Android优化技术详解

陈德春 编著

国内第一本系统讲解Android优化的作品

布局优化 + 内存优化 + 代码优化 + 性能优化 + 系统优化

循序渐进、讲解细致、写法通俗易懂，提供完善的售后技术支持。



清华大学出版社

Android 优化技术详解

陈德春　编　著

清华大学出版社
北　京

内 容 简 介

本书循序渐进而又详细地讲解了 Android 优化技术的基本知识。本书内容新颖、知识全面、讲解详细。全书分为 12 章，第 1 章讲解了 Android 系统的基础知识；第 2 章讲解了 Android 核心框架；第 3 章详细讲解了为什么要优化；第 4 章详细讲解了 UI 布局优化的基本知识；第 5 章详细讲解了 Android 内存系统的基本知识；第 6 章讲解了 Android 内存优化的基本知识；第 7 章讲解了代码优化的基本知识；第 8 章讲解了性能优化的基本知识；第 9 章讲解了系统优化的基本知识；第 10 章讲解了开发一个 Android 优化系统的基本知识；第 11 章和第 12 章是两个综合实例，分别讲解了在手机地图系统和 Android 足球游戏中使用优化技术的知识。书中的每个实例都遵循先提出制作思路及所包含知识点，在实例最后总结知识点，并让读者举一反三。

本书定位于 Android 的初、中级用户，既可作为初学者的参考书，也可作为有一定基础读者的提高书。

图书在版编目(CIP)数据

Android 优化技术详解/陈德春编著. --北京：清华大学出版社，2014
ISBN 978-7-302-35933-3

Ⅰ. ①A…　Ⅱ. ①陈…　Ⅲ. ①移动终端—应用程序—程序设计　Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字(2014)第 061856 号

责任编辑：李玉萍
装帧设计：杨玉兰
责任校对：王　晖
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn, 010-62791865
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张：**27.5　　**字　　数：**660 千字
版　　次：2014 年 4 月第 1 版　　**印　　次：**2014 年 4 月第 1 次印刷
印　　数：1～3000
定　　价：56.00 元

产品编号：048047-01

前　言

Foreword

进入 21 世纪以来，整个社会已经逐渐变得陌生了！生活和工作的快节奏令我们目不暇接，各种各样的信息充斥着我们的视野、撞击着我们的思维。追忆过去，Windows 操作系统的诞生成就了微软的霸主地位，也造就了 PC 时代的繁荣。然而，以 Android 和 iPhone 手机为代表的智能移动设备的发明却敲响了 PC 时代的警钟！移动互联网时代已经来临，谁会成为这些移动设备上的主宰？毫无疑问，这就是 Android——PC 时代的 Windows!

看 3G 的璀璨绚丽

随着 3G 的到来，无线带宽越来越高，使得在手机上布置更多内容丰富的应用程序成为可能，如视频通话、视频点播、移动互联网冲浪、在线看书/听歌、内容分享等。为了承载这些数据应用及快速部署，手机功能将会越来越智能，越来越开放。为了实现这些需求，必须有一个好的开发平台来支持，由 Google 公司发起的 OHA 联盟走在了业界的前列，2007 年 11 月推出了开放的 Android 平台，任何公司及个人都可以免费获取到源代码及开源 SDK。由于其开放性和优异性，Android 平台得到了业界广泛的支持，其中包括各大手机厂商和著名的移动运营商等。继 2008 年 9 月第一款基于 Android 平台的手机 G1 发布之后，预计三星、摩托罗拉、索爱、LG、华为等公司都将推出自 Gflg~Android 平台的手机，中国移动也将联合各手机厂商共同推出基于 Android 平台的 OPhone。按目前的发展态势，我们有理由相信，Android 平台能够在短时间内跻身智能手机开发平台的前列。

自从 2009 年 3G 牌照在国内发放后，3G、Android、iPhone、Google、苹果、手机软件、移动开发等词越来越充斥于耳。随着 3G 网络的大规模建设和智能手机的迅速普及，移动互联网时代已经微笑着迎面而来。

以创新的搜索引擎技术而一跃成为互联网巨头的 Google，无线搜索成为它进军移动互联网的一块基石。早在 2007 年，Google 中国就把无线搜索当作战略重心，不断推出新产品，尝试通过户外媒体推广移动搜索产品，并积极与运营商、终端厂商、浏览器厂商等达成战略合作。

Android 操作系统是 Google 最具杀伤力的武器之一。苹果以其天才的创新，使得 iPhone 在全球迅速拥有了数百万忠实“粉丝”，而 Android 作为第一个完整、开放、免费的手机平台，使开发者在为其开发程序时拥有更大的自由。与 Windows Mobile、Symbian 等厂商不同的是，Android 操作系统免费提供给开发人员，这样可节省近三成成本，因此得到了众多厂商与开发者的拥护。自从进入 2011 年后，Android 就一直是市场占有率最高的智能手机系统。并且 Android 的成功也造就了使用 Android 系统的手机制造商，现在三星借助 Android

操作系统，已经成为世界上发货量最大的手机制造商。

巨大的优势

从技术角度而言，Android 与 iPhone 相似，都采用 WebKit 浏览器引擎，具备触摸屏、高级图形显示和上网功能，用户能够在手机上查收电子邮件、搜索网址和观看视频节目等。Android 手机比 iPhone 等其他手机更强调搜索功能，界面更强大，可以说是一种融入了全部 Web 应用的平台。Android 的版本包括：Android 1.1、Android 1.5、Android 1.6、Android 2.0 等，当前的最新版本是 Android 4.2。随着版本的更新，从最初的触屏到现在的多点触摸，从普通的联系人到现在的数据同步，从简单的 GoogleMap 到现在的导航系统，从基本的网页浏览到现在的 HTML 5，这都说明 Android 已经逐渐稳定，而且功能越来越强大。此外，Android 平台不仅支持 Java、C、C++等主流的编程语言，还支持 Ruby、Python 等脚本语言，甚至 Google 还专为 Android 的应用开发推出了 Simple 语言，这使得 Android 有着非常广泛的开发群体。

优化的目的是提高用户体验

我们做任何一款产品，目标用户群体永远是消费者，而用户体验往往决定了一款产品的畅销程度。作为智能手机来说，因为其自身硬件远不及 PC，所以这就要求我们需要为消费者提供拥有更好用户体验的产品，只有这样我们的产品才会受追捧。

用户体验的英文是 User Experience，简称 UE。它是一种纯主观的在用户使用产品过程中建立起来的感受。但是对于一个界定明确的用户群体来讲，其用户体验的共性是能够经良好设计实验认识到。新竞争力在网络营销基础与实践中曾提到计算机技术和互联网的发展，使技术创新形态正在发生转变，以用户为中心、以人为本越来越得到重视，用户体验也因此被称作创新 2.0 模式的精髓。在中国面向知识社会的创新 2.0——应用创新园区模式探索，更将用户体验作为“三验”创新机制之首。

本书的内容

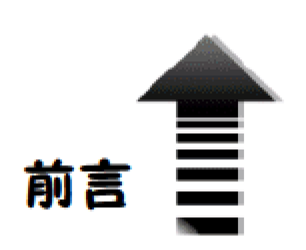
本书循序渐进地、详细地讲解了 Android 优化技术的基本知识，内容新颖、知识全面、讲解详细。Android 优化技术博大精深，需要程序员具备极高的水准和开发经验。笔者从事 Android 开发也是短短数载，也不可能完全掌握 Android 优化技术。本书将尽可能地将 Android 优化技术的核心内容展现给读者，书中主要提供了以下优化内容。

(1) UI 布局优化

讲解了优化 UI 界面布局的基本知识以及各种布局的技巧，剖析了减少层次结构、延迟加载和嵌套优化等方面的知识。

(2) 内存优化

详细讲解了 Android 系统内存的基本知识，分析了 Android 独有的垃圾回收机制，并分别剖析了缩放处理、数据保存、使用与释放、内存泄漏和内存溢出等方面的知识。



(3) 代码优化

讲解了在编码过程中，优化代码提高运行效率的基本知识。

(4) 性能优化

讲解了资源存储、加载 DEX 文件和 APK、虚拟机的性能、平台优化、优化渲染机制等方面的知识。

(5) 系统优化

详细讲解了进程管理器、设置界面、后台停止、转移内存程序和优化缓存等方面的知识。

(6) 优化工具

详细讲解了市面上常见的优化工具，例如优化大师、进程管理等。

科学的学习方法

不要认为学习 Android 技术是一件很困难的事情，不断寻找规律，学习新知识和新技能，积累经验，这几乎是每一个电脑高手必经的成长之路。中国有句古话“授人以鱼，不如授人以渔”，说的是传授给人既有知识，不如传授给人学习知识的方法。通过本书，我们将告诉读者学习的方法，并介绍一条比较清晰的学习之路。

1. 积极的心态

无论是知识还是技能，智者之所以能够更好、更快地掌握知识和技能，很大程度上得益于良好的学习方法。人们常说：兴趣是最好的老师，压力是前进的动力，要想获得一个积极的心态，最好能对学习对象保持浓厚的兴趣。如果暂时提不起兴趣，那么就重视来自工作或生活的压力，把它们转化为学习的动力。

2. 注重实践

读者在学习本书的过程中，建议学完理论后，进行实际操作。首先学习书中的理论，再动手调试本书中的实例，然后用模拟器运行书中的例子，只有这样才能做到印象深刻，才能真正理解 Android 优化技术的基本知识。这样在实际应用中遇到其他类似问题时，才能做到熟能生巧、触类旁通。

3. 善用资源，学以致用

对于计算机优化技术，除了少部分专业人士外，大部分人学习的目的是为了应用，通过优化技术解决工作中的问题并提高工作效率。“解决问题”常常是促使人学习的一大动机，带着问题学习，不但进步快，而且很容易对优化技术产生更大的兴趣，从而获得持续的进步。

(1) 善用资源

在学习过程中，难免会遇到自己不理解的知识，此时可以找一些相关的书籍来阅读，不断尝试解决问题。或者通过互联网的搜索引擎找到问题的解决办法，善用搜索引擎，基本上可以找到大多数问题所在！

(2) QQ 群

如果在互联网上找不到问题的解决办法，可以通过 QQ 访问相关学习群，群中的高手们会对你提出的问题进行回答。

(3) 向优化技术高手学习

在练习实际操作能力时，可以虚心向优化技术领域的高手学习。如果读者闭门造车，盲人摸象，则很难掌握技术精髓。而经过身边的优化技术高手指点，可以轻松掌握相关的技能。

本书特色

本书内容相当丰富，覆盖面全，涉及了 Android 优化技术人员成长道路上的方方面面。我们的目标是通过一本图书，提供多本图书的价值，读者可以根据自己的需要有选择地阅读，以完善本人的知识和技能。在内容的编写上，本书具有以下特色。

(1) 结构合理

从用户的实际需要出发，科学安排知识结构，内容由浅入深，叙述清楚，并附有相应的总结和练习，具有很强的知识性和实用性，反映了当前 Android 优化技术的发展和应用水平。同时全书精心筛选的最具代表性、读者最关心的知识点，几乎包括了 Android 优化技术的所有方面。

(2) 易学易懂

本书条理清晰、语言简洁，可帮助读者快速掌握每个知识点；每个部分既相互连贯又自成体系，使读者既可以按照本书编排的章节顺序进行学习，也可以根据自己的需求对某一章节进行针对性的学习。

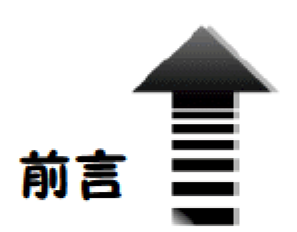
(3) 实用性强

本书彻底摒弃枯燥的理论和简单的操作，注重实用性和可操作性，将 Android 优化技术的理论融合到实际的操作环境中，使用户掌握相关操作技能的同时，还能学习到相应的开发知识。

本书的读者对象

本书在内容安排上由浅入深，写作上层层剥洋葱式的分解，充分实例举证，非常适合 Android 开发技术的初学者，同时也适合具有一定 Android 开发基础，想对 Android 开发技术进一步了解和掌握的中级用户。如果你是以下类型的读者，本书会带领你迅速进入 Android 开发领域。

- 有一定 Android 开发经验的读者。
- 从事 Android 开发的研究人员和工作人员。
- 有一定的 Android 基础，想快速学会 Android 高级技术的读者。
- 有一定 Android 开发基础，需要加深对 Android 技术核心进一步了解和掌握的程序员。
- 高等院校相关专业的学生，或需要编写论文的学生。



- 企业和公司在职人员、需要提高学习或工作需要的程序员。
- 从事 Android 移动网络开发等相关工作的技术人员。

本书由陈德春编著，其他参与本书编写的人员还有：王石磊、周秀、付松柏、邓才兵、钟世礼、谭贞军、王东华、罗红仙、王振丽、熊斌。由于本人水平有限，纰漏和不尽如人意之处在所难免，诚请读者提出宝贵意见或建议，以便修订并使之更臻完善。售后 QQ：729017304。

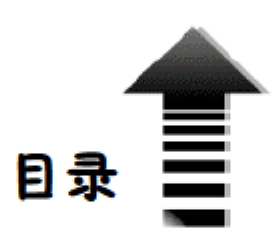
本书部分源代码网络下载路径：http://www.tup.tsinghua.edu.cn。

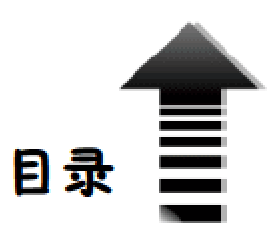
编　者



目　录

Contents





目录

Android

第1章

Android 系统闪亮登场

Android 是谷歌公司于 2007 年推出的一款智能手机平台，它是建立在 Linux 的开源内核基础之上的，能够迅速建立手机软件的解决方案。虽然 Android 的外形比较简单，但是其功能十分强大，当前已经成为一个新兴的热点，是市场占有率排名第一的智能手机操作系统。本章将简单介绍 Android 系统的相关知识，让读者了解 Android 系统的发展之路。

1.1 一款全新的智能手机平台——Android

其实在 Android 系统诞生之前，智能手机就已经存在了，并且受到了广大消费者的追捧。随着人们日益对手机功能的追求，各大手机厂商纷纷建立了自己的智能手机操作系统来满足消费者的需求。Android 手机系统就是在这个背景下诞生的，并且一经推出，便迅速发展，成为当前最受欢迎的智能手机操作系统之一。

1.1.1 何谓智能手机

智能手机是指具有像个人电脑那样强大的功能，拥有独立的操作系统，可以在上面安装我们需要的游戏等第三应用程序。在 Android 系统诞生之前，市面上已经存在多款智能手机产品，例如诺基亚的 Symbian 系列和微软的 Windows Mobile 系列等。

一般来说，智能手机必须具备下面的功能标准。

(1) 在上面可以安装一些新的应用软件。

(2) 具有高速处理数据的芯片。

(3) 可以播放音乐和视频。

(4) 具有大存储芯片和存储扩展能力。

(5) 支持 GPS 导航。

手机联盟公布的智能手机的主要特点如下。

(1) 具备普通手机的所有功能，例如，可以进行正常的通话和收发短信等基本的手机应用。

(2) 是一个开放性的操作系统，在系统上可以安装更多的应用程序，从而实现功能的无限扩充。

(3) 具备上网功能。

(4) 具备 PDA 的功能，能够实现个人信息管理、日程记事、任务安排、多媒体应用、浏览网页等。

(5) 可以根据个人需要扩展机器的功能。

(6) 扩展性能强，并且可以支持很多第三方软件。

1.1.2 看当前主流的智能手机系统

当今市面上最主流的智能手机系统当属微软、塞班、PDA、黑莓、苹果和本书的主角——Android。

1. 微软的 Windows Mobile

Windows Mobile 是微软公司的一款杰出产品，它将熟悉的 Windows 桌面扩展到了个人设备中。使用 Windows Mobile 操作系统的设备主要有 PPC 手机、PDA、随身音乐播放器等。Windows Mobile 操作系统有三种，分别是 Windows Mobile Standard、Windows



Mobile Professional 和 Windows Mobile Classic。目前市面上常见的版本是 Windows Phone 7 和 Windows Phone 8。

2. 塞班系统 Symbian

塞班系统出自由诺基亚、索尼爱立信、摩托罗拉、西门子等几家大型移动通信设备商共同出资组建的一个合资公司，专门研发手机操作系统，现已被诺基亚全额收购。Symbian 有着良好的界面，采用内核与界面分离技术，对硬件的要求比较低，支持 C++、Visual Basic 和 J2ME。目前根据人机界面的不同，Symbian 体系的 UI(User Interface，用户界面)平台分为 Series 60、Series 80、Series 90、UIQ 等。其中 Series 60 主要是给数字键盘手机使用，Series 80 是为完整键盘所设计，Series 90 则是为触控笔方式而设计。

注意： ① 2010 年 9 月，诺基亚宣布将从 2011 年 4 月起从 Symbian 基金会(Symbian Foundation)手中收回 Symbian 操作系统控制权。由此看来，诺基亚在 2008 年全资收购塞班公司之后，希望继续扩大塞班影响力的愿望并没有实现。
② 在苹果和 Android 的强大市场攻势下，诺基亚在 2011 年 2 月 11 日宣布与微软达成广泛战略合作关系，并将 Windows Phone 作为其主要的智能手机操作系统。这家芬兰手机巨头试图通过结盟扭转颓势。截至本书成稿时，诺基亚和微软联合推出了最新版本 Windows Phone 8。

3. Palm

Palm 是流行的个人数字助理(PDA，又称掌上电脑)的传统名字。从广义上讲，Palm 是 PDA 的一种，是 Palm 公司发明的。而从狭义上讲，Palm 是 Palm 公司生产的 PDA 产品，区别于 SONY 公司的 Clie 和 Handspring 公司的 Visor/Treo 等其他运行 Palm 操作系统的 PDA 产品。其显著特点之一是写入装置输入数据的方法，能够点击显示器上的图标选择输入的项目。2009 年 2 月 11 日，Palm 公司 CEO Ed Colligan 宣布，以后将专注于 WebOS 和 Windows Mobile 的智能设备，而将不会再有基于“Palm OS”的智能设备推出，除了 Palm Centro 会在以后和其他运营商合作时继续推出。

4. 黑莓 BlackBerry

BlackBerry 是加拿大 RIM 公司推出的一种移动电子邮件系统终端，其特色是支持推动式电子邮件、手提电话、文字短信、互联网传真、网页浏览及其他无线资讯服务，其最大的优势是收发邮件。正因为这一优势，所以受到了商务用户的青睐。

5. iOS

iOS 作为苹果移动设备 iPhone 和 iPad 的操作系统，在 App Store 的推动之下，成为世界上引领潮流的操作系统之一。原来这个系统名为“iPhone OS”，直到 2010 年 6 月 7 日 WWDC 大会上宣布改名为“iOS”。对 iOS 用户界面的控制方法包括滑动、轻触开关及按键；与系统交互功能包括滑动(Swiping)、轻按(Tapping)、挤压(Pinching，通常用于缩小)及反向挤压(Reverse Pinching or Unpinching，通常用于放大)。此外通过其自带的加速器，可以令其旋转设备改变其 y 轴以令屏幕改变方向，这样的设计令 iPhone 更易于使用。

从最初的 iPhone OS，演变至最新的 iOS 系统，iOS 成为苹果新的移动设备操作系统，

横跨 iPod Touch、iPad、iPhone，成为苹果最强大的操作系统。甚至新一代的电脑系统——Mac OS 系列也借鉴了 iOS 系统的一些设计，可以说 iOS 是苹果的又一个成功的操作系统，能给用户带来极佳的使用体验。

6. Android

Android 是本书的主角，是谷歌于 2007 年 11 月 5 日宣布的基于 Linux 内核的开源手机操作系统的名称。Android 平台由操作系统、中间件、用户界面和应用软件组成，号称是首个为移动终端打造的真正开放和完整的移动软件。

注意： 2011 年 8 月 15 日，谷歌和摩托罗拉移动公司共同宣布，谷歌将以每股 40.00 美元现金收购摩托罗拉移动，总额约 125 亿美元，相比摩托罗拉移动股份的收盘价溢价了 63%，双方董事会都已全票通过该交易。谷歌 CEO 拉里·佩奇表示，摩托罗拉移动将完全专注于 Android 系统，收购摩托罗拉移动之后，将增强整个 Android 生态系统。佩奇同时表示，Android 将继续开源，收购的一个目的是为了获得专利。

1.2　分析 Android 的优势

从 2007 年 11 月 5 日诞生之日起，到 2011 年 7 月，Android 系统在智能手机中的占有率高达 43%，位居智能手机系统占有率排行榜的第一位。并且随着各大厂商新产品的推出，必然会继续巩固这一地位。为什么 Android 能在这么多的智能系统中脱颖而出，成为市场占有率第一的手机系统呢？经分析，总结出如下四个优势，是它吸引厂商和消费者青睐的原因。

1.2.1　第一个优势——出自名门

Android 出身于 Linux 家族，是一款号称开源的手机操作系统。当 Android“一炮走红”之后，各大手机联盟纷纷加入，并且都推出了各自系列产品。这个联盟由包括中国移动、三星、摩托罗拉、高通、宏达电子和 T-Mobile 等在内的 30 多家技术和无线应用的领军企业组成。通过与运营商、设备制造商、开发商和其他有关各方结成深层次的合作伙伴关系，希望借助建立标准化、开放式的移动电话软件平台，在移动产业内形成一个开放式的生态系统。

1.2.2　第二个优势——强大的开发团队

Android 的研发队伍阵容豪华，包括摩托罗拉、Google、HTC(宏达电子)、PHILIPS、T-Mobile、高通、魅族、三星、LG 以及中国移动在内的 34 家企业，它们都将基于该平台开发手机的新型业务，应用之间的通用性和互联性将在最大程度上得到保持，并且还成立了手机开放联盟，联盟中包括世界各国手机制造业中的巨头和通信巨头。

1.2.3　第三个优势——奖金丰厚

谷歌为了提高程序员的开发积极性，不但为它们提供了一流硬件的设置和一流的软件服务，而且还采取了振奋人心的奖励机制，定期举行比赛，创意和应用夺魁者将会得到重奖。

1.2.4　第四个优势——代码开源

开源意味着对开发人员和手机厂商来说，Android 是完全无偿、免费使用的。由于源代码公开的原因，所以吸引了全世界各地无数程序员的热情。于是很多手机厂商都纷纷采用 Android 作为自己产品的系统，甚至包括很多山寨厂商。而对于开发人员来说，众多厂商的采用就意味着人才需求大，所以纷纷加入到 Android 开发大军中来。于是有一些干的还可以的程序员禁不住高薪的诱惑，都纷纷改行做 Android 开发。至于“混”的不尽如人意的程序员，就更加坚定了“改行做 Android 手机开发”，目的是想寻找自己程序员生涯的转机。并且有很多遇到发展瓶颈的程序员，也决定做 Android 开发，因为这样可以学习一门新技术，使自己的未来更加有保障。

1.3　搭建开发环境

对于 Android 开发人员来说，在进行开发之前首先需要搭建一个对应的开发环境。在具体搭建 Android 开发环境之前，需要先明确了解 Android 开发包括以下两个部分。

(1) 底层开发：大多数是指和硬件相关的开发，并且是基于 Linux 环境的，例如开发驱动程序。

(2) 应用开发：是指开发能在 Android 系统上运行的程序，例如游戏、地图等程序。本书的重点是讲解游戏应用开发，即使讲一些底层的知识，也是为上层的应用服务的。

另外，因为当前市面中最主流的操作系统是 Windows，所以本书只介绍在 Windows 环境下搭建 Android 开发环境的过程。

1.3.1　安装 Android SDK 的系统要求

在搭建开发环境之前，一定先确定基于 Android 应用软件所需要开发环境的要求，具体如表 1-1 所示。

表 1-1　开发系统所需求环境

项　目	版本要求	说　明	备　注
操作系统	Windows XP/Windows 7/Windows 8	根据自己的电脑配置自行选择	选择自己最熟悉的操作系统



项　目	版本要求	说　明	备　注
软件开发包	Android SDK	选择最新版本的 SDK	截至发稿，最新手机版本是 4.1，最普及的版本是 2.3
IDE	Eclipse IDE+ADT	Eclipse 3.3 以上版本和 ADT(Android Development Tools)开发插件	选择 for Java Developer
其他	JDK Apache Ant	Java SE Development Kit 5 或 6	不能选择单独的 JRE 进行安装，必须要有 JDK

Android 开发工具是由多个开发包组成的，其中最主要的开发包如下。

- JDK：可以到网址 http://www.oracle.com/technetwork/java/javase/downloads/index.html 下载。
- Eclipse：可以到网址 http://www.eclipse.org/downloads/下载 Eclipse IDE for Java Developers。
- Android SDK：可以到网址 http://developer.android.com 下载。
- 下载对应的开发插件。

1.3.2　安装 JDK、Eclipse、Android SDK

本书介绍的安装是以 Windows 7 为平台，安装的软件为 JDK 1.6、Eclipse 3.3、ADT 1.5、Android SDK 4.0。下面具体介绍各自的安装步骤，并且在提供的网络资源中有详细的介绍。

1. 安装 JDK

安装 Eclipse 的开发环境需要 JRE 的支持，在 Windows 上安装 JRE/JDK 非常简单，其流程如下。

(1) 在 Oracle 官方网站下载，网址为 http://www.oracle.com/technetwork/java/javase/downloads/index.html，如图 1-1 所示。

(2) 在图 1-1 中可以看到有很多版本，运行 Eclipse 时虽然只需要 JRE 就可以了，但是在开发 Android 应用程序的时候，需要完整的 JDK(JDK 已经包含了 JRE)，且要求其版本在 1.5 以上，这里选择 Java SE (JDK) 6，其下载页面如图 1-2 所示。

(3) 在图 1-2 中找到 JDK 6 Update 22，单击其右侧的 Download 按钮后弹出填写登录信息界面，在此输入你的账号信息，如果没有账号可以免费注册一个，然后单击 Continue 按钮，如图 1-3 所示。

(4) 来到选择操作系统和语言界面，在此选择 Windows 选项，然后单击 Download 按钮，如图 1-4 所示。

经过上述操作后，开始下载安装文件 jdk-6u22-windows-i586.exe。

(5) 下载完成后双击 jdk-6u22-windows-i586.exe 开始进行安装，将弹出安装向导对话框，在此单击【下一步】按钮，如图 1-5 所示。

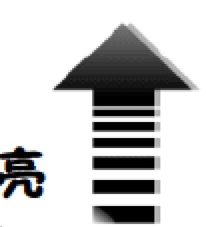






图 1-1　Oracle 官方下载页面

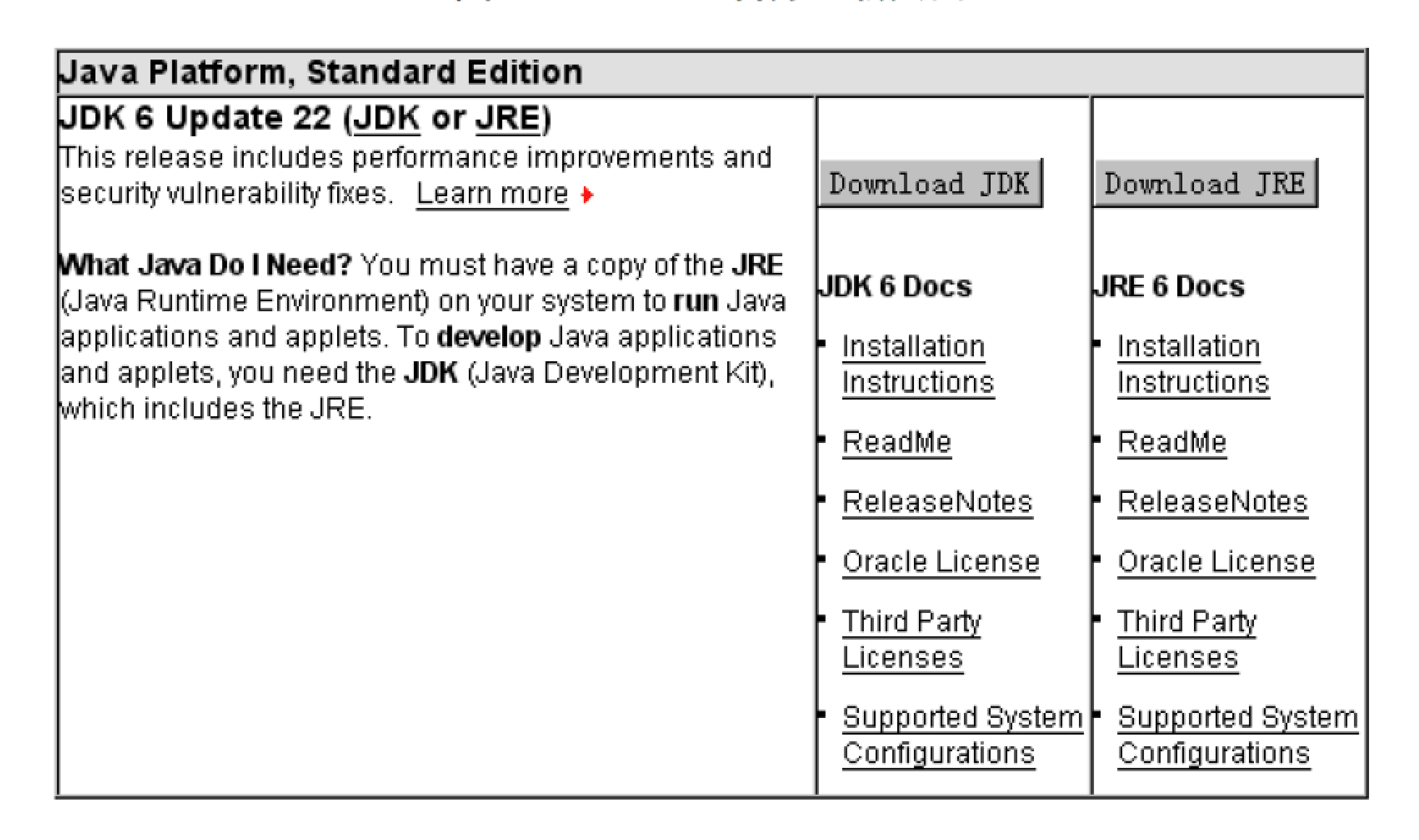


图 1-2　JDK 下载页面

(6) 弹出【自定义安装】界面，在此选择文件的安装路径，如图 1-6 所示。

(7) 单击【下一步】按钮，开始进行安装，如图 1-7 所示。

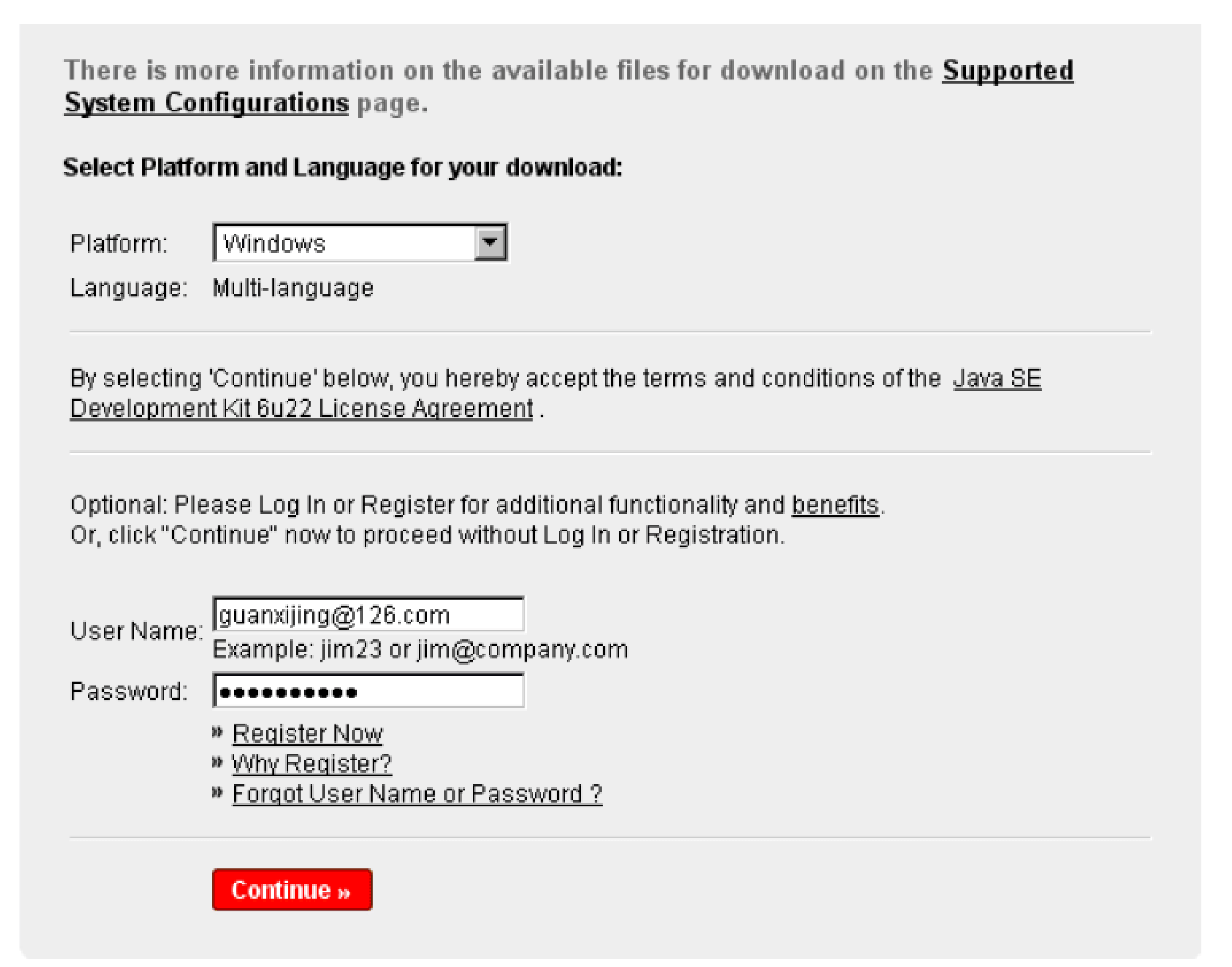


图 1-3　输入账号信息

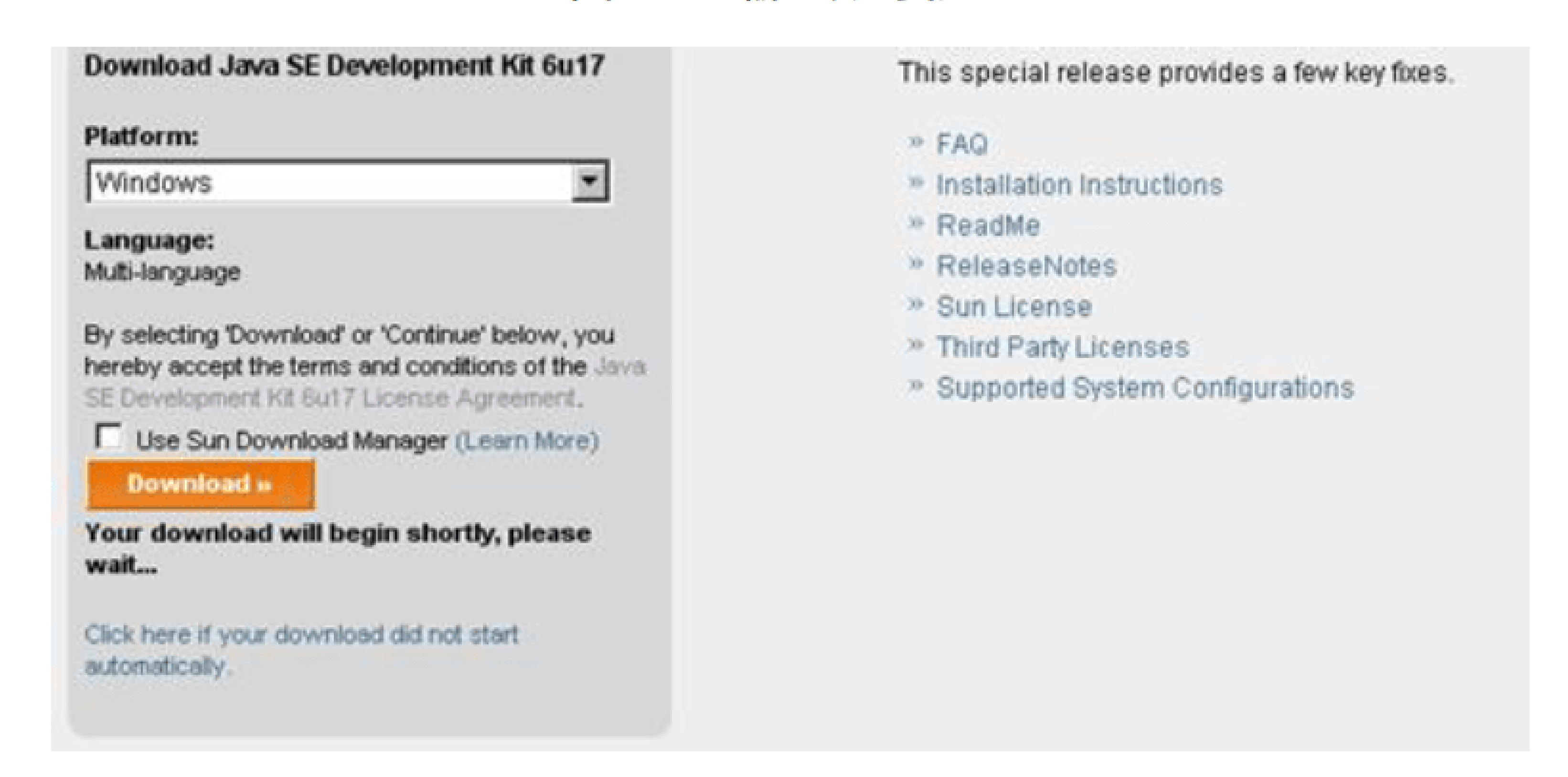


图 1-4　选择 Windows 选项



图 1-5　安装向导对话框

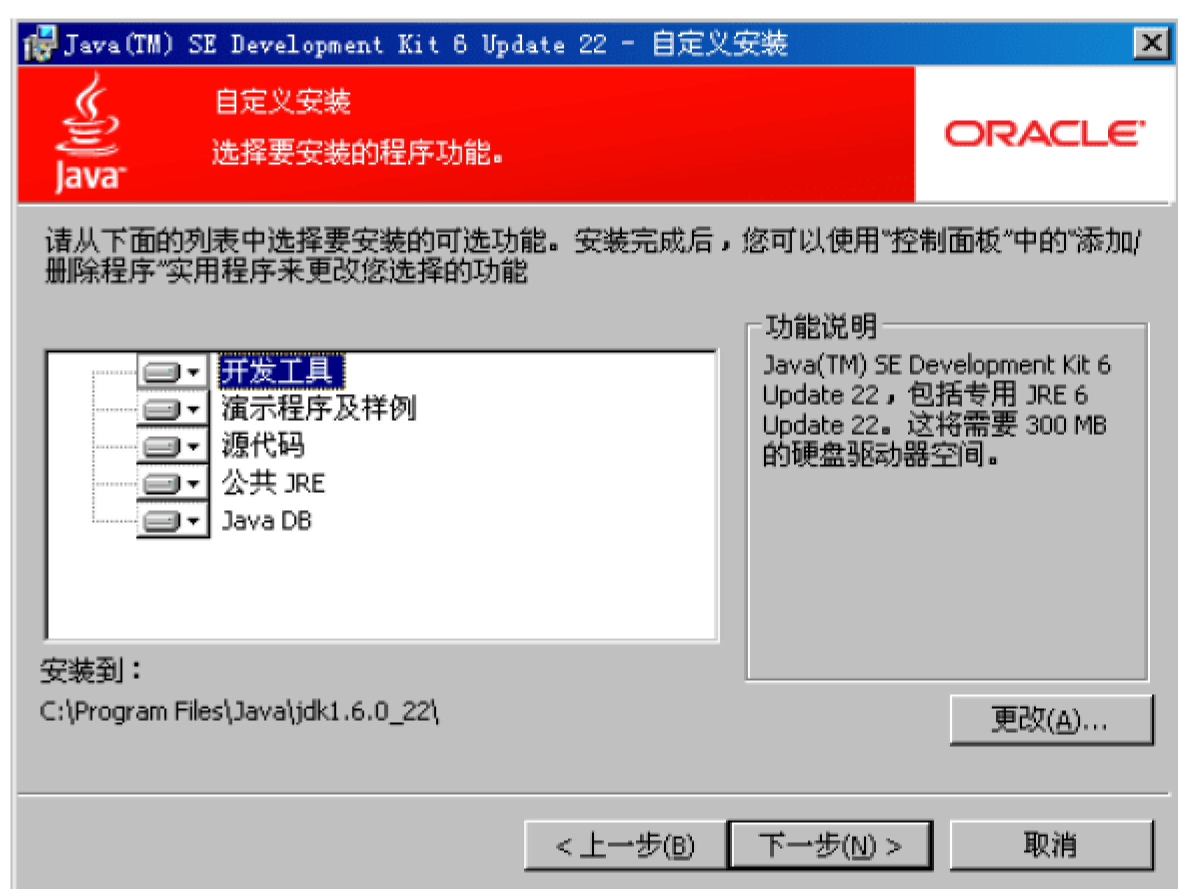


图 1-6　选择安装路径



图 1-7　开始安装

(8) 完成后弹出【目标文件夹】界面，在此选择要安装的位置，如图 1-8 所示。



图 1-8　【目标文件夹】界面

(9) 单击【下一步】按钮后继续安装，如图 1-9 所示。



图 1-9　继续安装

(10) 完成后弹出【完成】对话框，单击【完成】按钮后完成整个安装过程，如图 1-10 所示。



图 1-10　完成安装

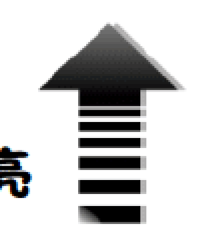
注意： 完成安装后可以检测是否安装成功，方法是依次选择【开始】|【运行】命令，在运行框中输入“cmd”并按下 Enter 键，在打开的 CMD 窗口中输入“java –version”，如果显示如图 1-11 所示的提示信息，则说明安装成功。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>java -version
java version "1.6.0_22"
Java(TM) SE Runtime Environment (build 1.6.0_22-b04)
Java HotSpot(TM) Client VM (build 17.1-b03, mixed mode, sharing)

C:\Documents and Settings\Administrator>
```

图 1-11　CMD 窗口



如果检测结果是没有安装成功，则需要将 Java 安装目录的绝对路径添加到系统的 PATH 中，具体做法如下。

(1) 右击【我的电脑】，在弹出的快捷菜单中选择【属性】命令，弹出【系统属性】对话框。切换到【高级】选项卡，单击下面的【环境变量】按钮，在弹出的【环境变量】对话框的【系统变量】选项组中单击【新建】按钮。在弹出的【新建系统变量】对话框的【变量名】文本框中输入“JAVA_HOME”，在【变量值】文本框中输入刚才的目录，比如笔者的是“C:\Program Files\Java\jdk1.6.0_22”，如图 1-12 所示。

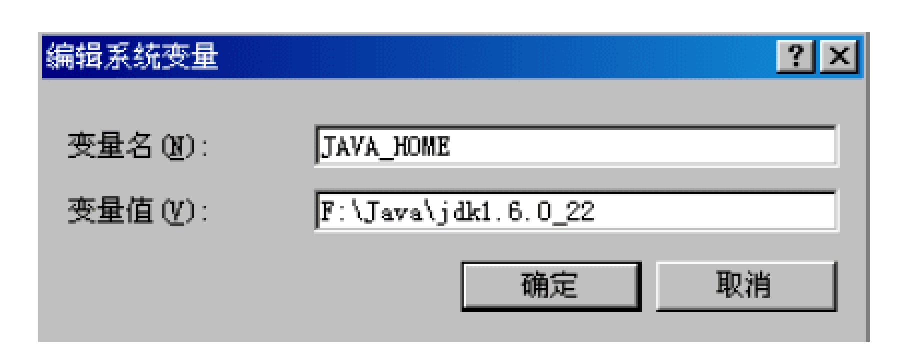


图 1-12　设置系统变量

(2) 再次新建一个变量名 classpath，其变量值如下。

```
.;%JAVA_HOME%/lib/rt.jar;%JAVA_HOME%/lib/tools.jar
```

单击【确定】按钮找到 PATH 的变量，双击或单击编辑，在变量值的最前面添加如下值。

```
%JAVA_HOME%/bin;
```

如图 1-13 所示。

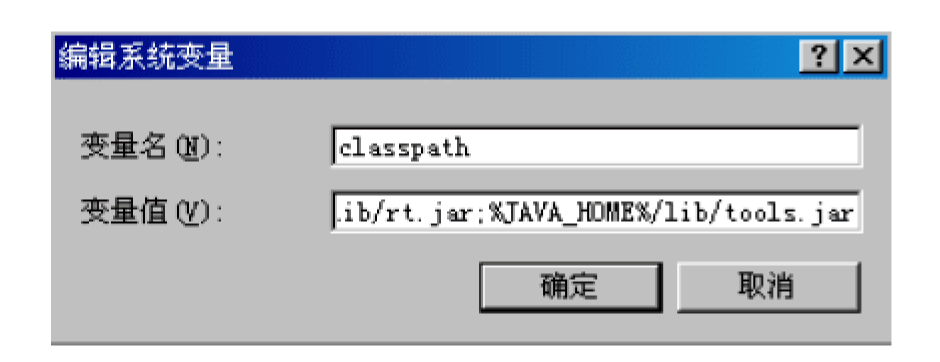


图 1-13　设置系统变量

(3) 依次选择【开始】|【运行】菜单命令，在【运行】对话框中输入“cmd”并按下 Enter 键，在打开的 CMD 窗口中输入“java –version”，如果显示如图 1-14 所示的提示信息，则说明安装成功。

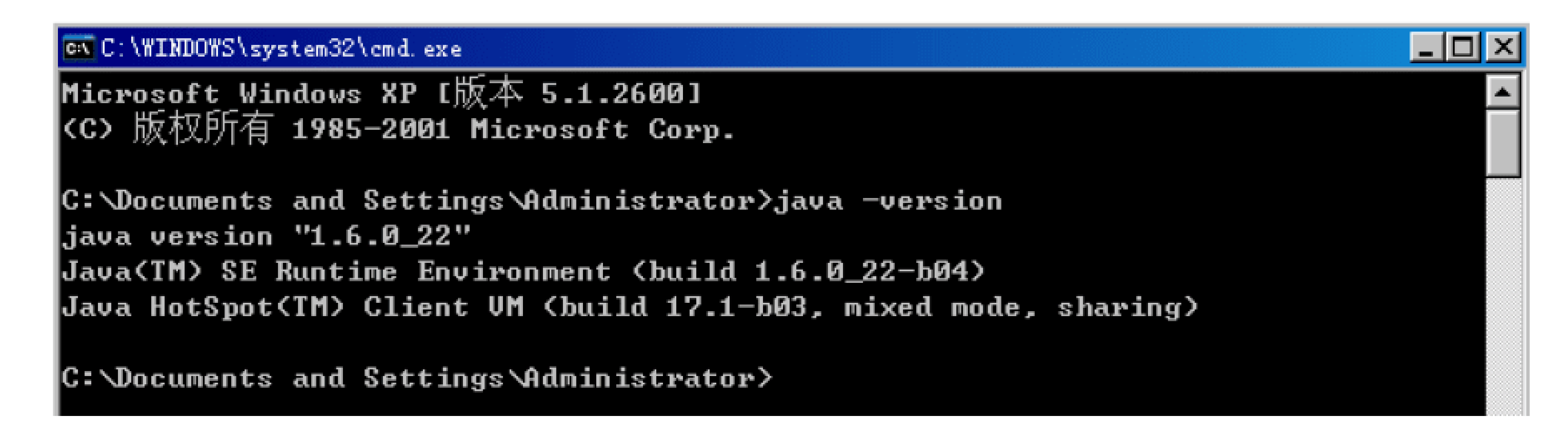


图 1-14　CMD 窗口

注意： 上述变量是按照笔者本人的安装路径设置的，笔者安装 JDK 的路径是 C:\Program Files\Java\jdk1.6.0_22。

2．安装 Eclipse

在安装好 JDK 后，就可以安装 Eclipse 了，具体步骤如下。

(1) 打开 Eclipse 的官方下载页面 http://www.eclipse.org/downloads/，如图 1-15 所示。

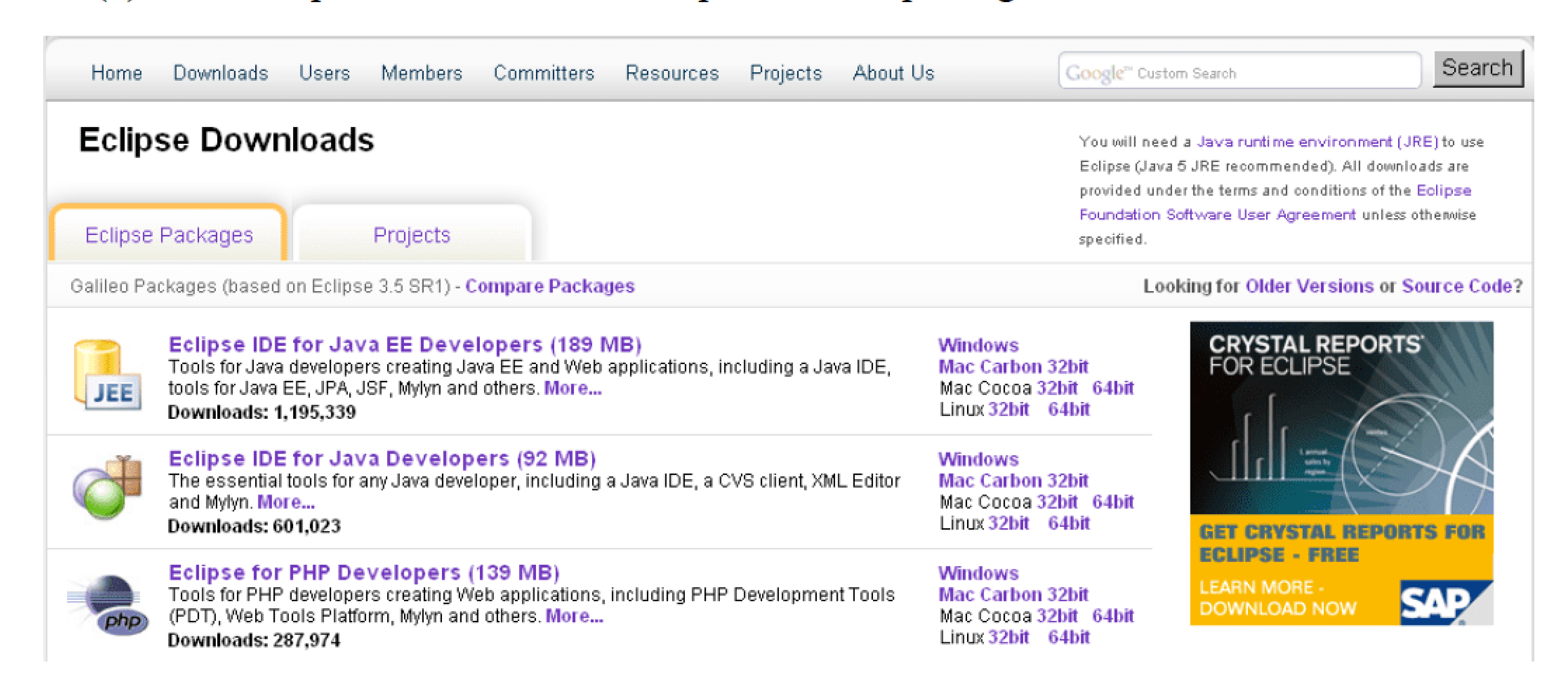


图 1-15　Eclipse 官方下载页面

(2) 在图 1-15 所示的界面中选择 Eclipse IDE for Java Developers (92 MB)，来到其下载的镜像页面，在此只需选择离用户最近的镜像即可(可以选择推荐的下载速度)，如图 1-16 所示。

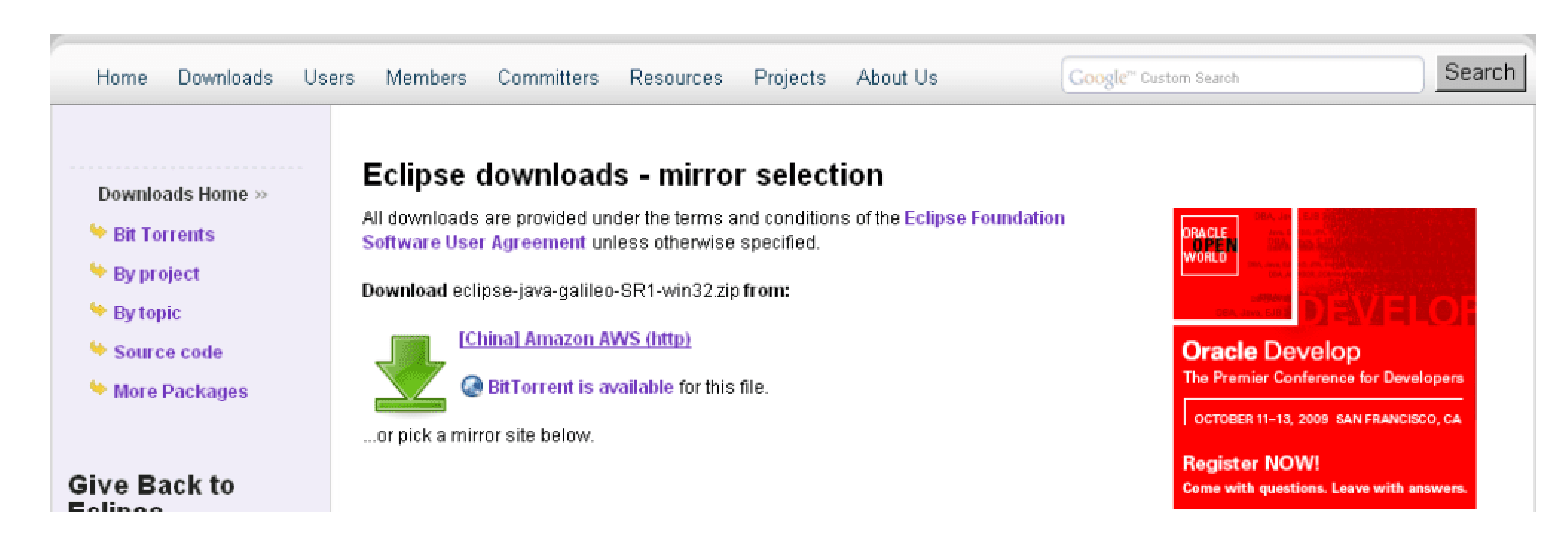


图 1-16　选择镜像

(3) 下载完成后，先找到下载的压缩包 eclipse-java-galileo-SR1-win32.zip。

注意： 解压 Eclipse 下载的压缩文件后就可以使用，而无须执行安装程序，不过在使用前一定要先安装 JDK。在此假设 Eclipse 解压后存放的目录为 F:\eclipse。

(4) 进入解压后的目录，此时可以看到一个名为“eclipse.exe”的可执行文件，双击此文件直接运行，Eclipse 能够自动找到我们先前安装的 JDK 路径。启动界面如图 1-17 所示。

(5) 如果是安装后第一次启动 Eclipse，会看到选择工作空间的界面提示，如图 1-18 所示。此时单击 OK 按钮，即可完成 Eclipse 的安装。

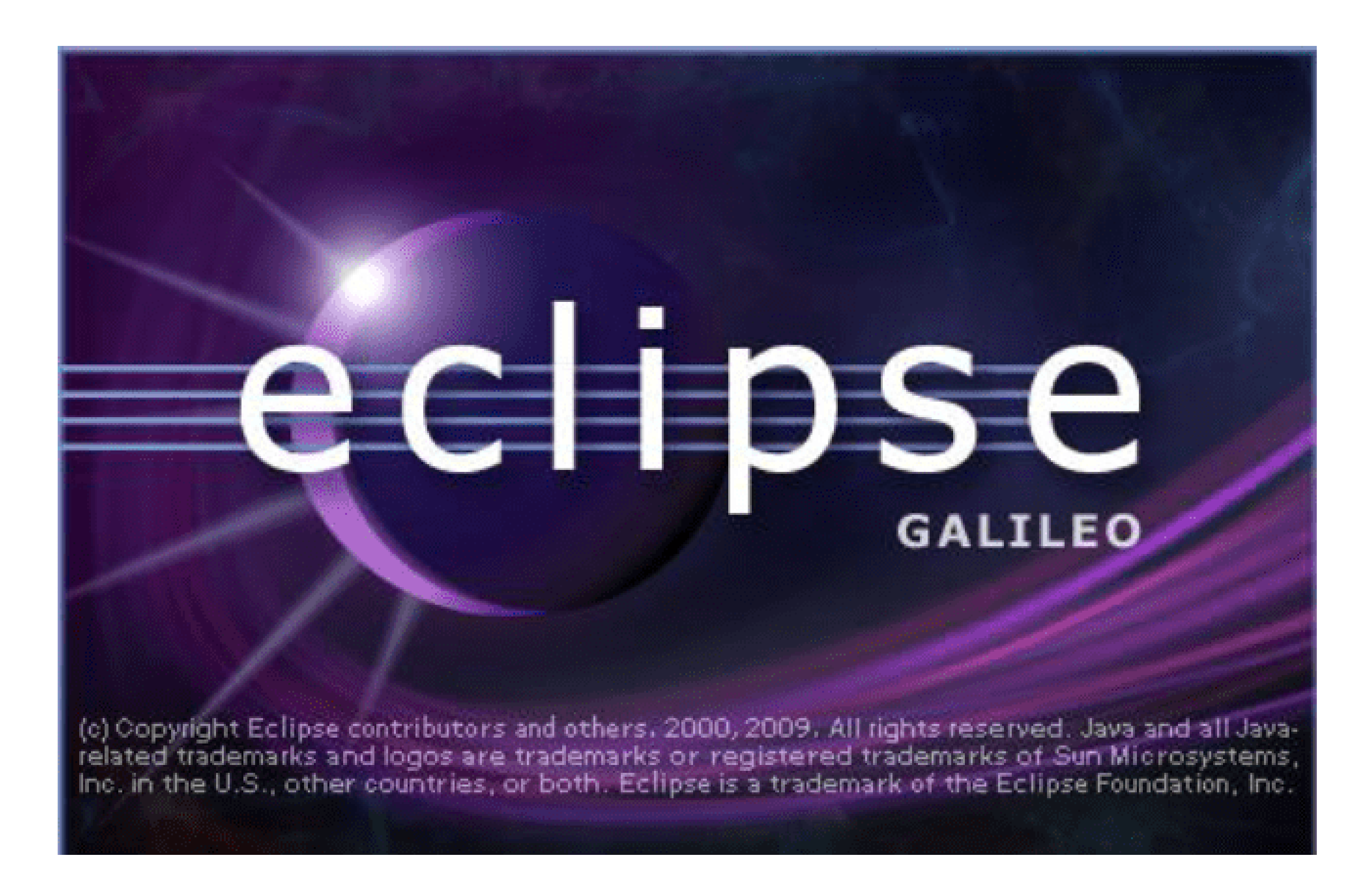


图 1-17　Eclipse 启动界面

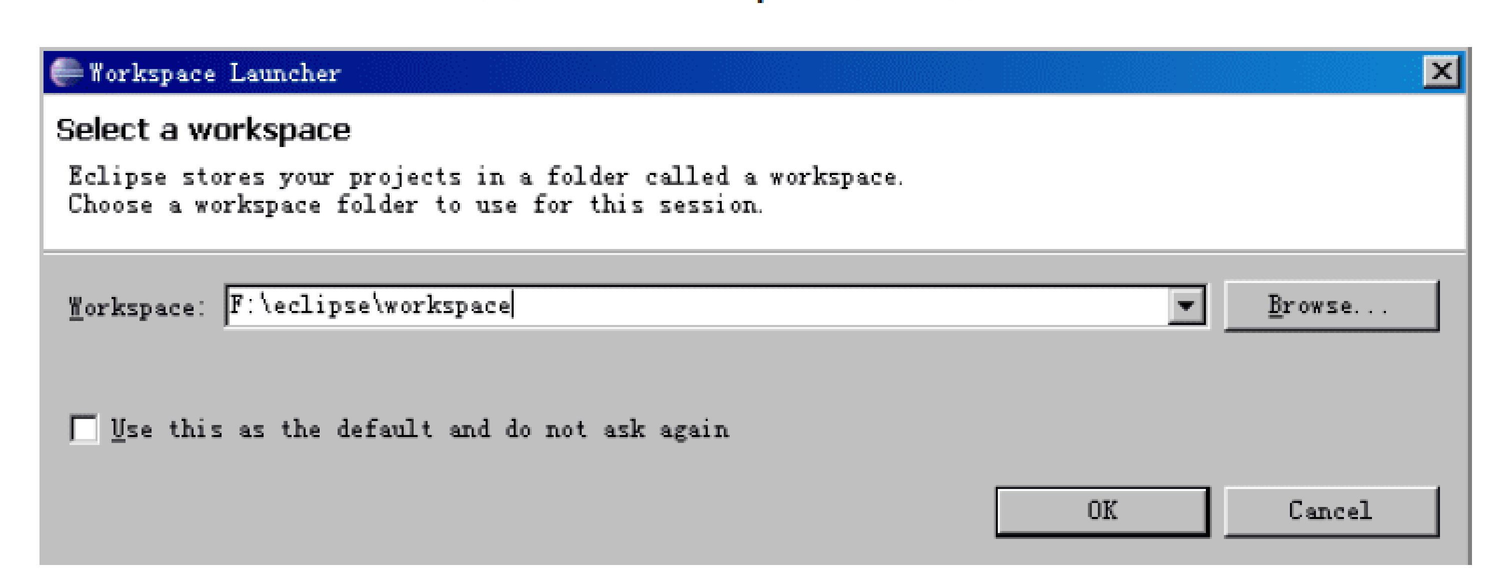
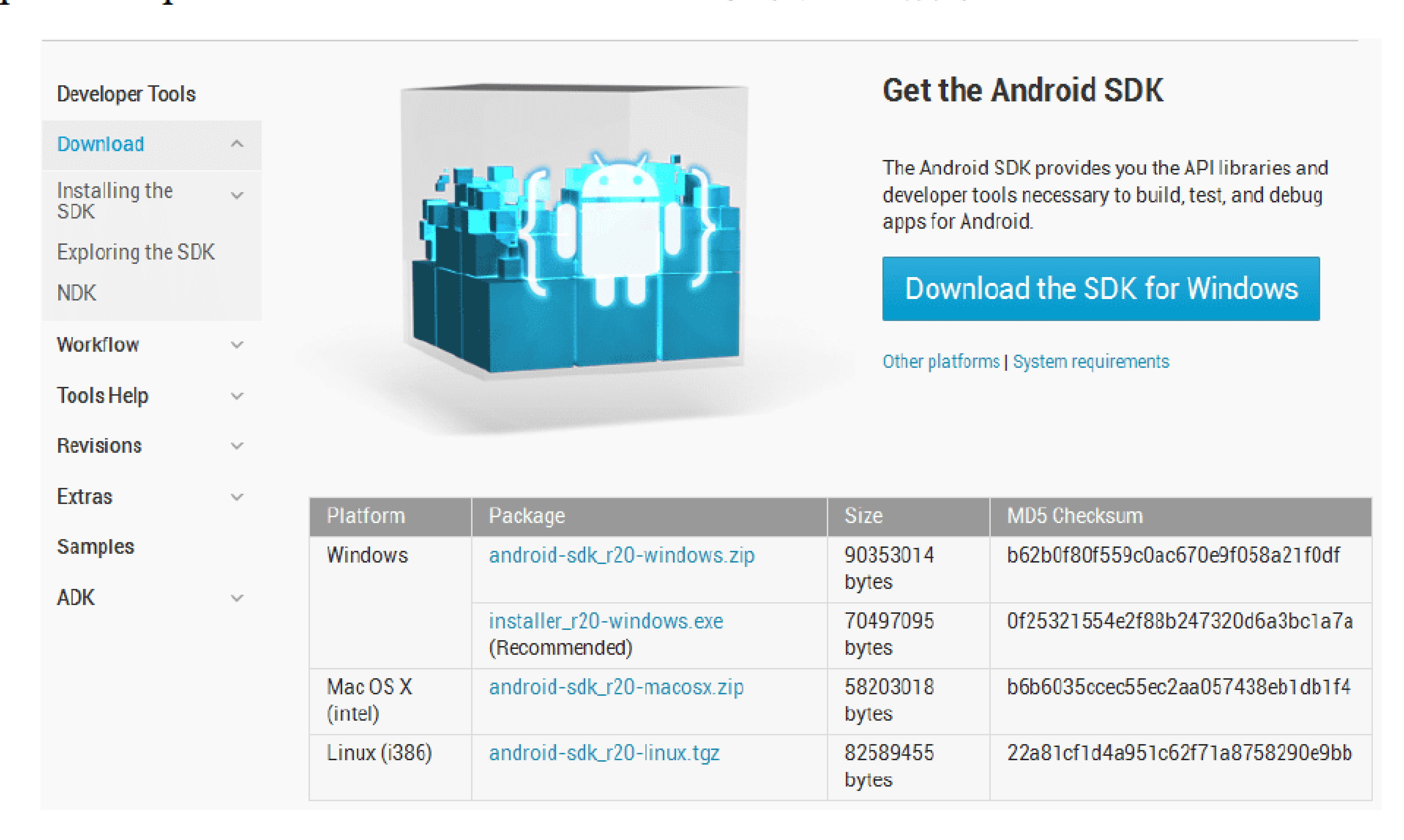


图 1-18　选择工作空间

3．安装 Android SDK

安装 JDK 和 Eclipse 后，接下来需要下载并安装 Android SDK，具体步骤如下。

(1) 打开 Android 开发者社区网址 http://developer.android.com/，然后转到 SDK 下载页面 http://developer.android.com/sdk/index.html，如图 1-19 所示。



Platform	Package	Size	MD5 Checksum
Windows	android-sdk_r20-windows.zip	90353014 bytes	b62b0f80f559c0ac670e9f058a21f0df
	installer_r20-windows.exe (Recommended)	70497095 bytes	0f25321554e2f88b247320d6a3bc1a7a
Mac OS X (intel)	android-sdk_r20-macosx.zip	58203018 bytes	b6b6035ccec55ec2aa057438eb1db1f4
Linux (i386)	android-sdk_r20-linux.tgz	82589455 bytes	22a81cf1d4a951c62f71a8758290e9bb

图 1-19　SDK 下载页面

(2) 在此选择用于 Windows 平台的链接 android-sdk_r20-windows.zip，弹出如图 1-20 所示的对话框。

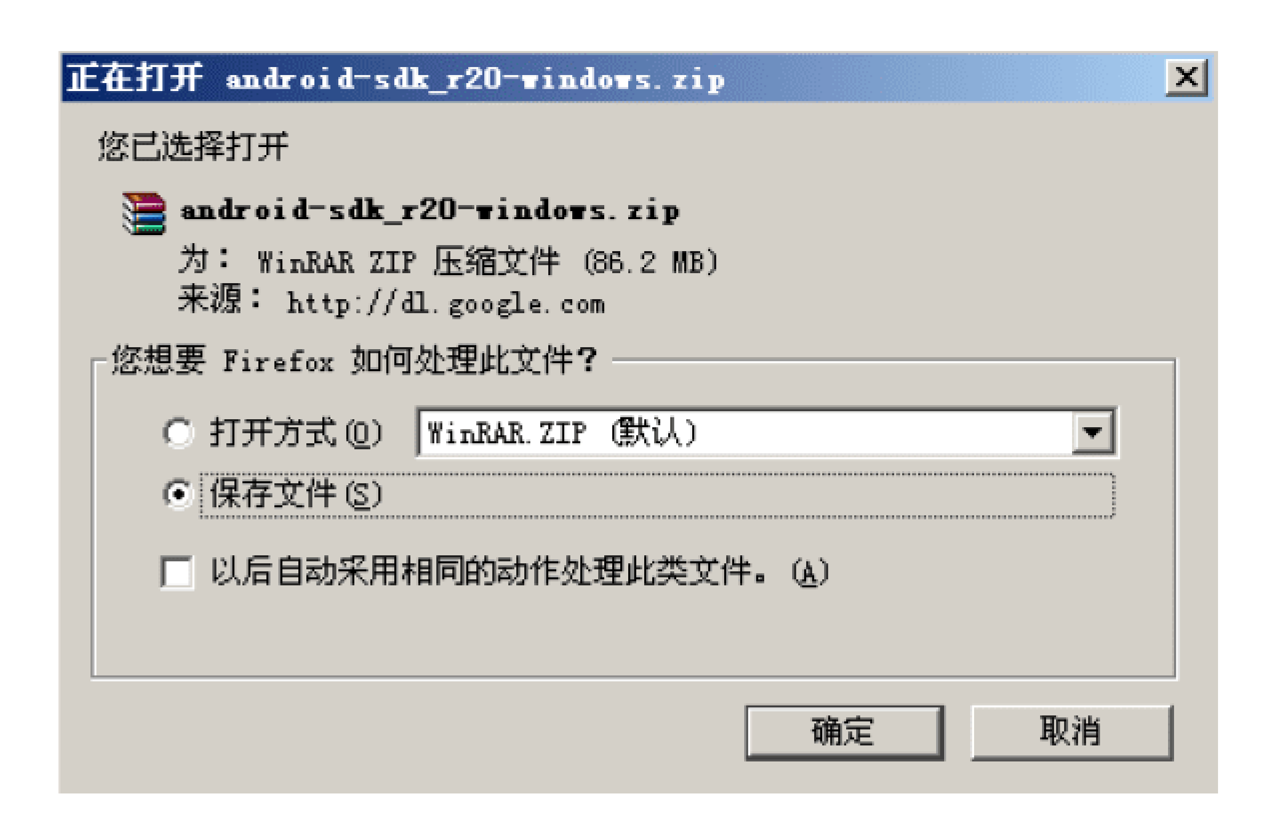


图 1-20　打开文件对话框

下载后解压压缩文件。假设下载后的文件解压存放在 F:\android\目录下，并将其 tools 目录的绝对路径添加到系统的 PATH 中，具体操作步骤如下。

(1) 右击【我的电脑】，在弹出的快捷菜单中选择【属性】命令，弹出【系统属性】对话框，切换到【高级】选项卡，单击下面的【环境变量】按钮，在弹出的【环境变量】对话框的【系统变量】选项组中单击【新建】按钮，在弹出的【新建系统变量】对话框的【变量名】文本框中输入“SDK_HOME”，在【变量值】文本框中输入刚才的安装目录，比如笔者的是 F:\android-sdk-windows，如图 1-21 所示。

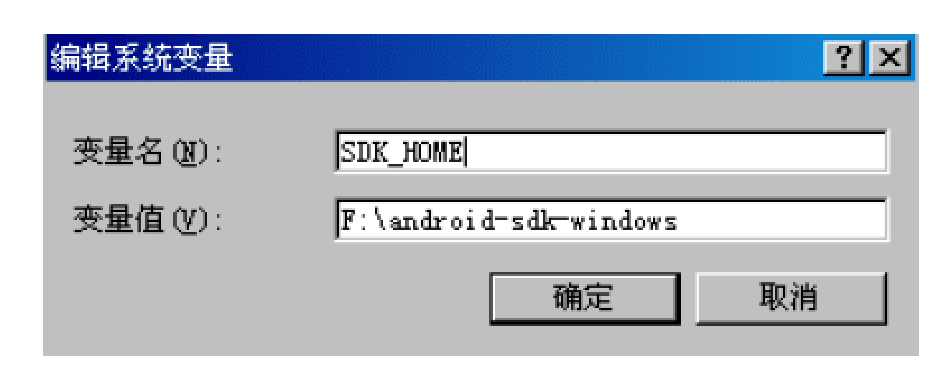


图 1-21　设置系统变量

(2) 找到 PATH 的变量，双击或单击编辑，在变量值最前面加上“%SDK_HOME%\tools;”，如图 1-22 所示。

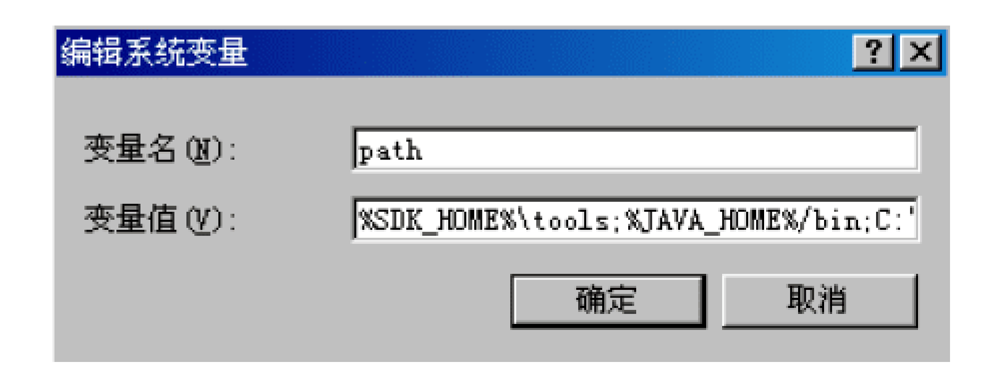


图 1-22　编辑系统变量

(3) 选择【开始】|【运行】菜单命令，在【运行】对话框中输入“cmd”并按下 Enter 键，在打开的 CMD 窗口中输入一个测试命令，例如 android -h，如果显示如图 1-23 所示的提示信息，则说明安装成功。



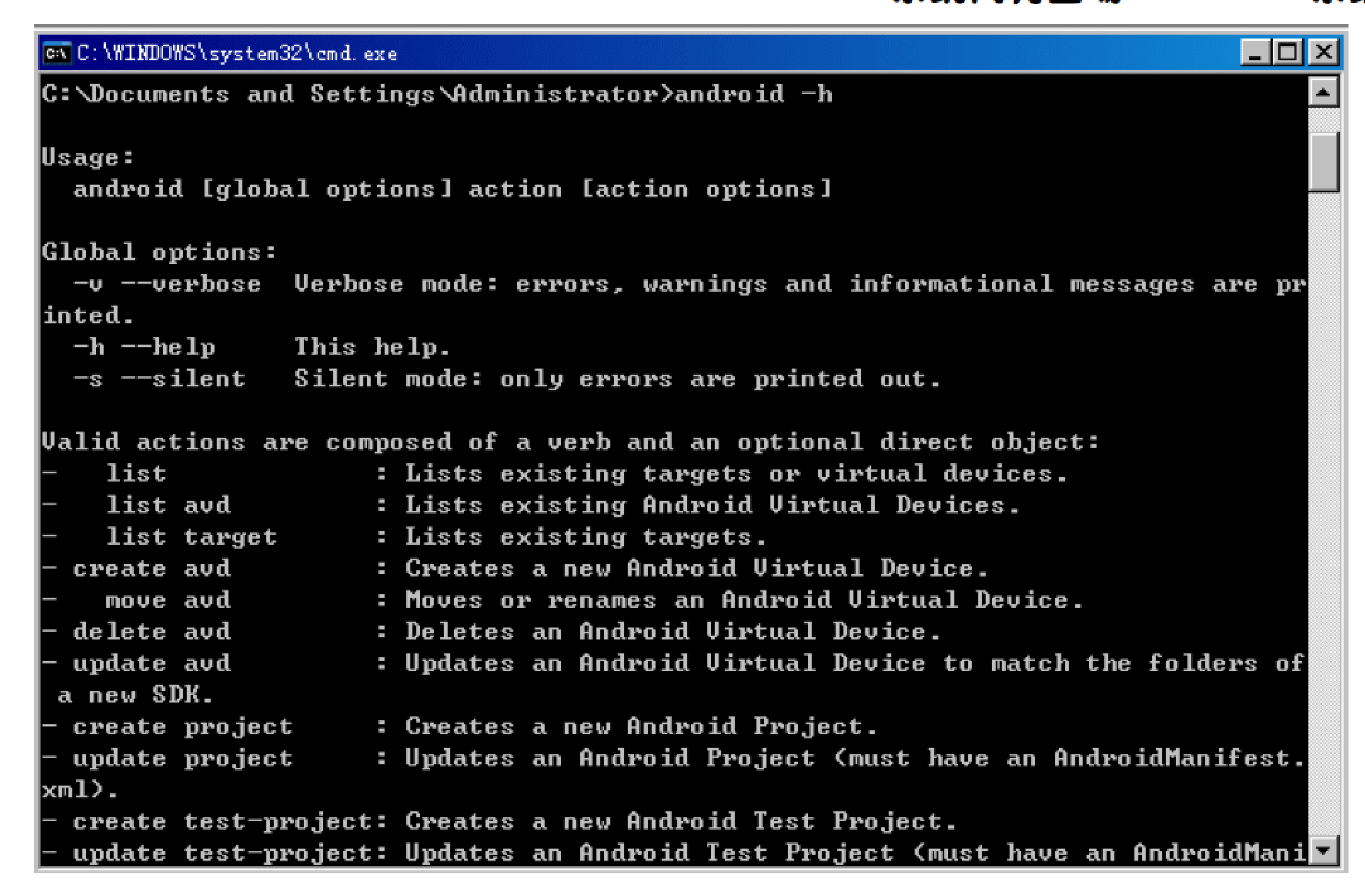


图 1-23　设置系统变量

4．安装 ADT

Android 为 Eclipse 定制了一个专用插件 Android Development Tools，简称 ADT，此插件为我们提供了一个开发 Android 应用程序的综合环境。ADT 扩展了 Eclipse 的功能，可以让用户快速地建立 Android 项目，创建应用程序界面。要安装 Android Development Tools plug-in，需要首先打开 Eclipse IDE，然后进行如下操作。

(1) 打开 Eclipse 后，依次选择菜单栏中的 Help | Install New Software 命令，如图 1-24 所示。

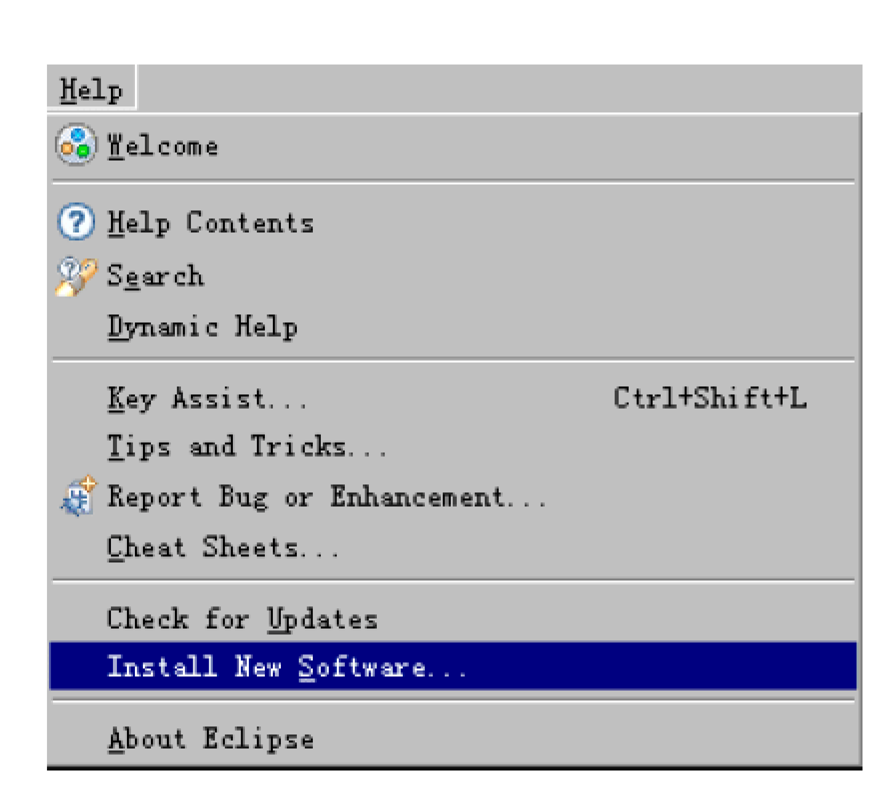


图 1-24　添加插件

(2) 在弹出的 Install 对话框中单击 Add 按钮，如图 1-25 所示。

(3) 在弹出的 Add Site 对话框中分别输入名字和地址，名字可以自己命名，例如 123，但是在 Location 文本框中必须输入插件的网络地址：http://dl-ssl.google.com/Android/eclipse/，如图 1-26 所示。



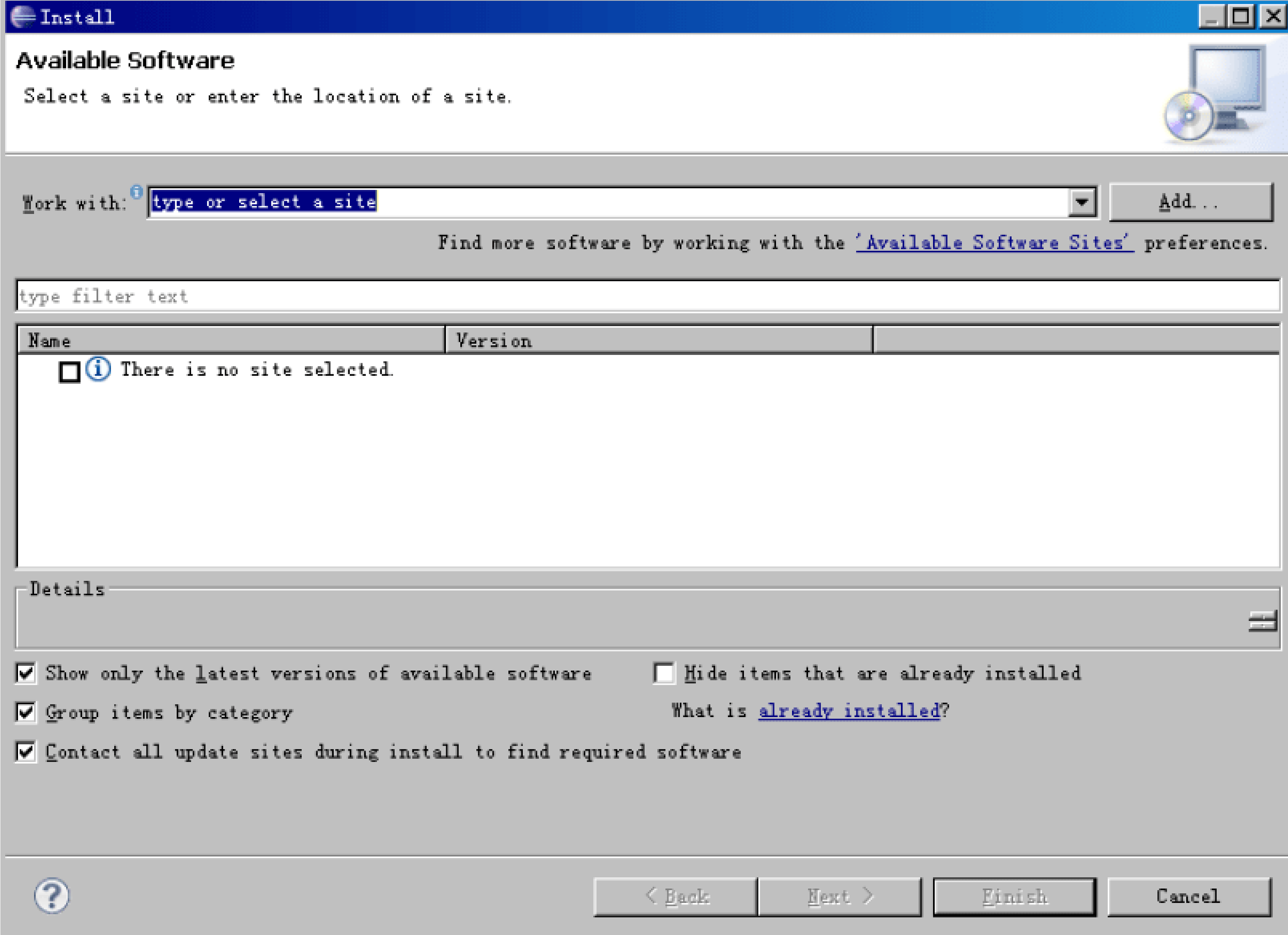


图 1-25　添加插件

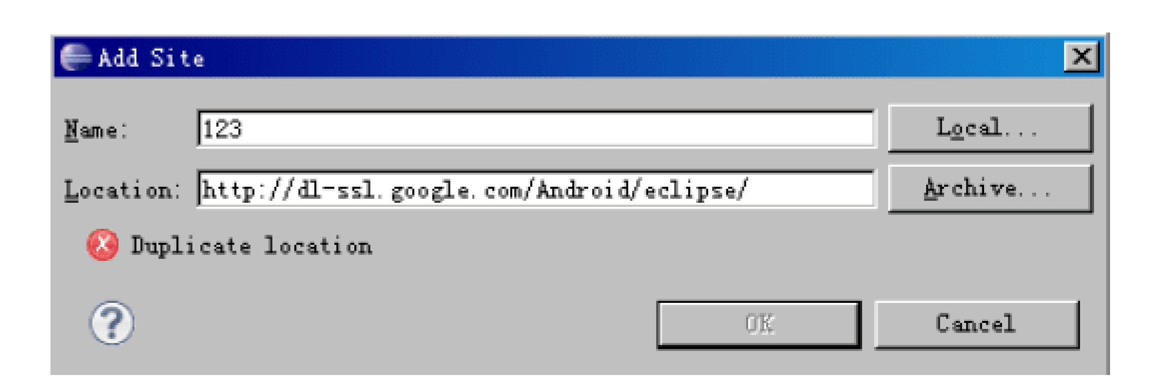


图 1-26　设置地址

(4) 单击 OK 按钮，此时在 Install 对话框中会显示系统中的可用插件，如图 1-27 所示。

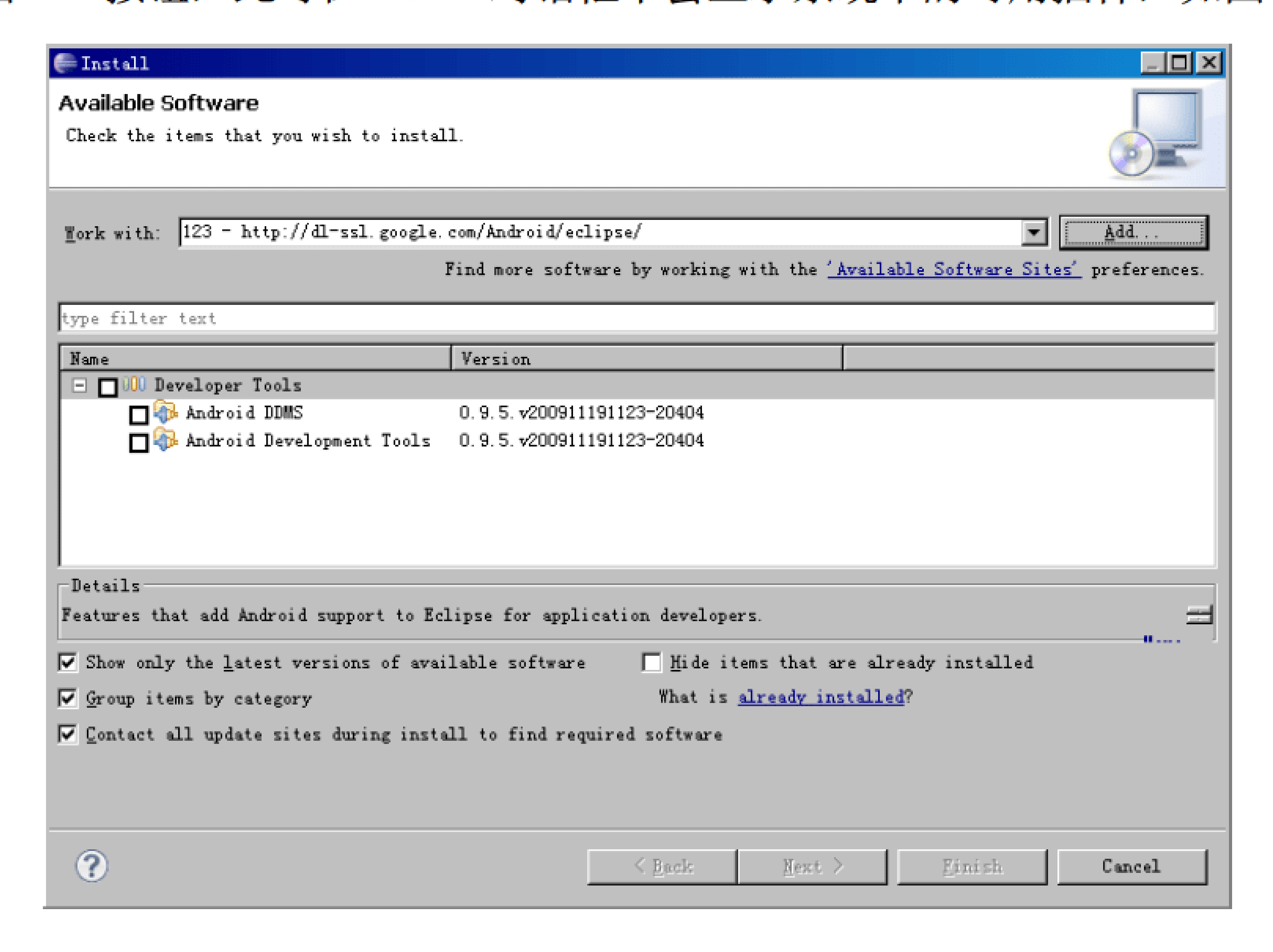


图 1-27　插件列表

(5) 选中 Android DDMS 和 Android Development Tools，然后单击 Next 按钮进入安装界面，如图 1-28 所示。

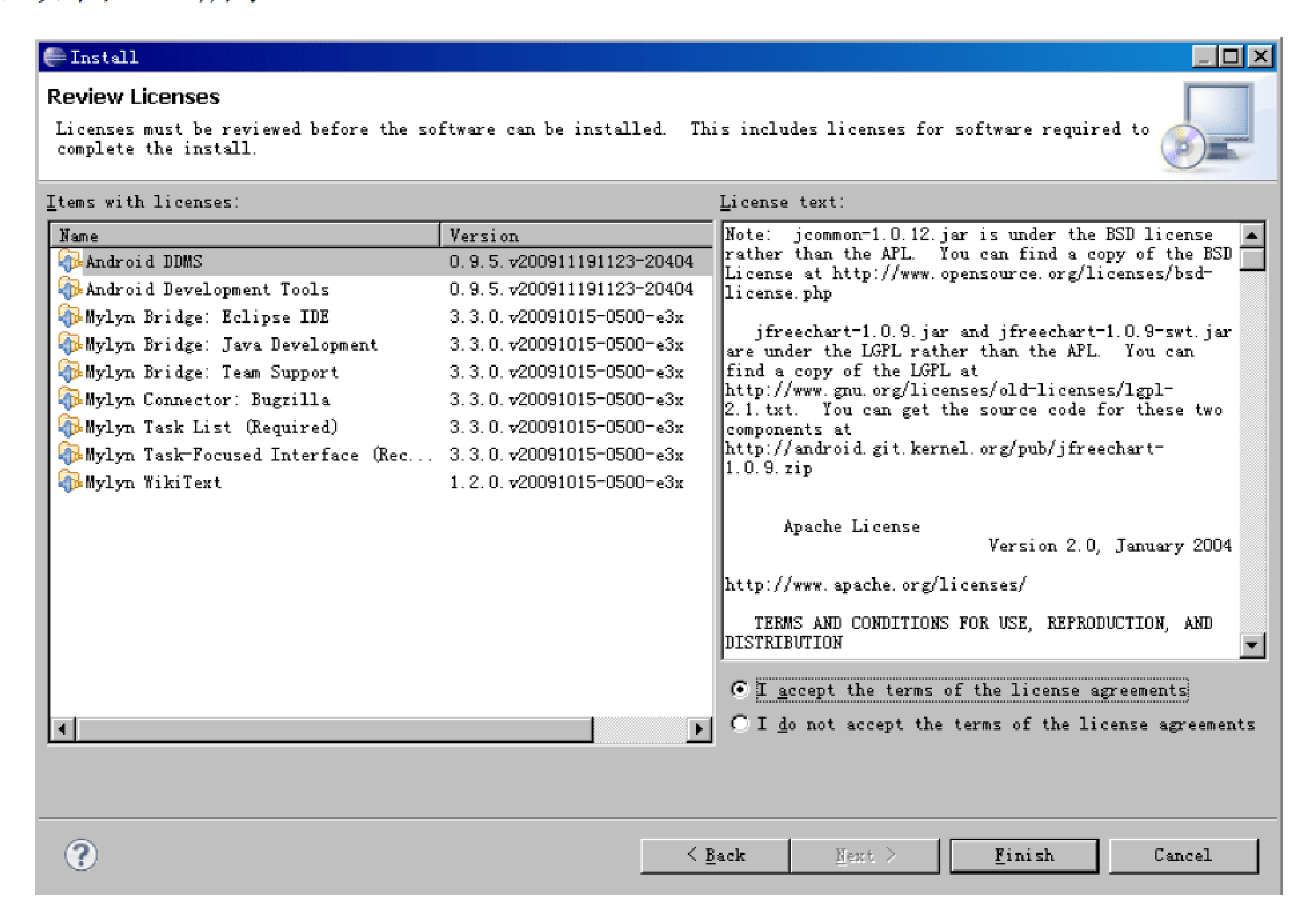


图 1-28　插件安装界面

(6) 选中 I accept the terms of the license agreements 单选按钮，然后单击 Finish 按钮后开始安装工作，如图 1-29 所示。

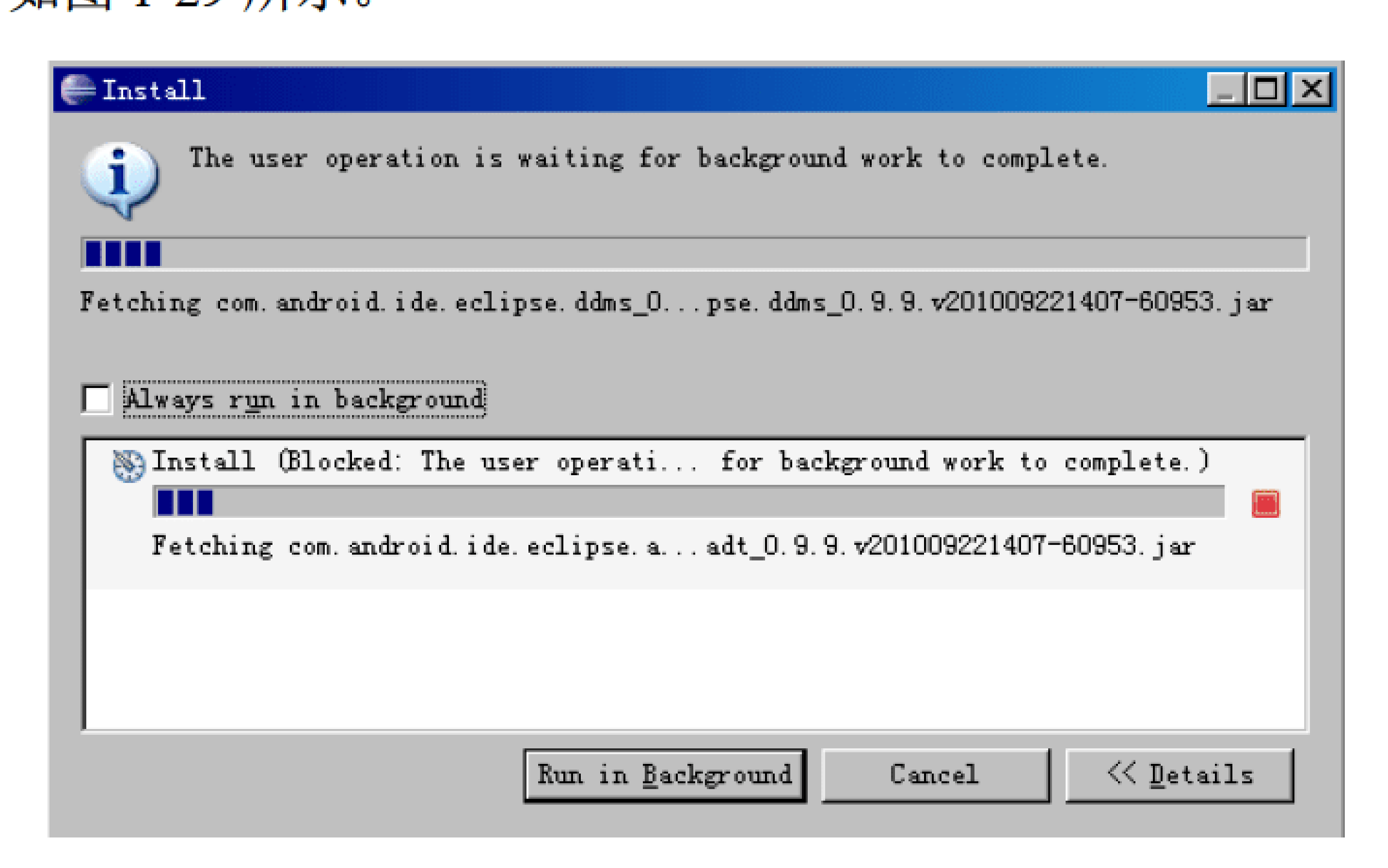


图 1-29　开始安装

在图 1-29 对应的步骤中，可能会发生“计算插件占用资源”的情况，整个计算过程有点慢。完成后会提示重启 Eclipse 来加载插件，重启之后就可以使用了。并且不同版本的 Eclipse 安装插件的方法和步骤是不同的，但是都大同小异，读者可以根据操作提示自行解决。

1.3.3 设定 Android SDK Home

当完成上述插件安装工作后，此时还不能使用 Eclipse 创建 Android 项目，我们还需要在 Eclipse 中设置 Android SDK 的主目录。

(1) 打开 Eclipse，在菜单中依次选择 Windows | Preferences 命令，如图 1-30 所示。

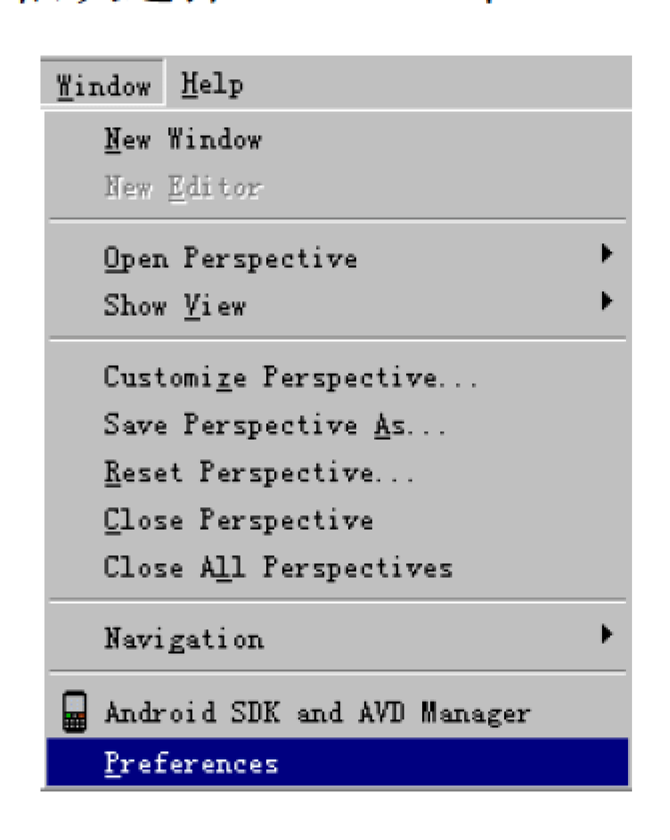


图 1-30 选择 Preferences 命令

(2) 在弹出的对话框左侧可以看到 Android 选项，选中该选项，在右侧设定 Android SDK 所在目录为 SDK Location，然后单击 OK 按钮完成设置，如图 1-31 所示。

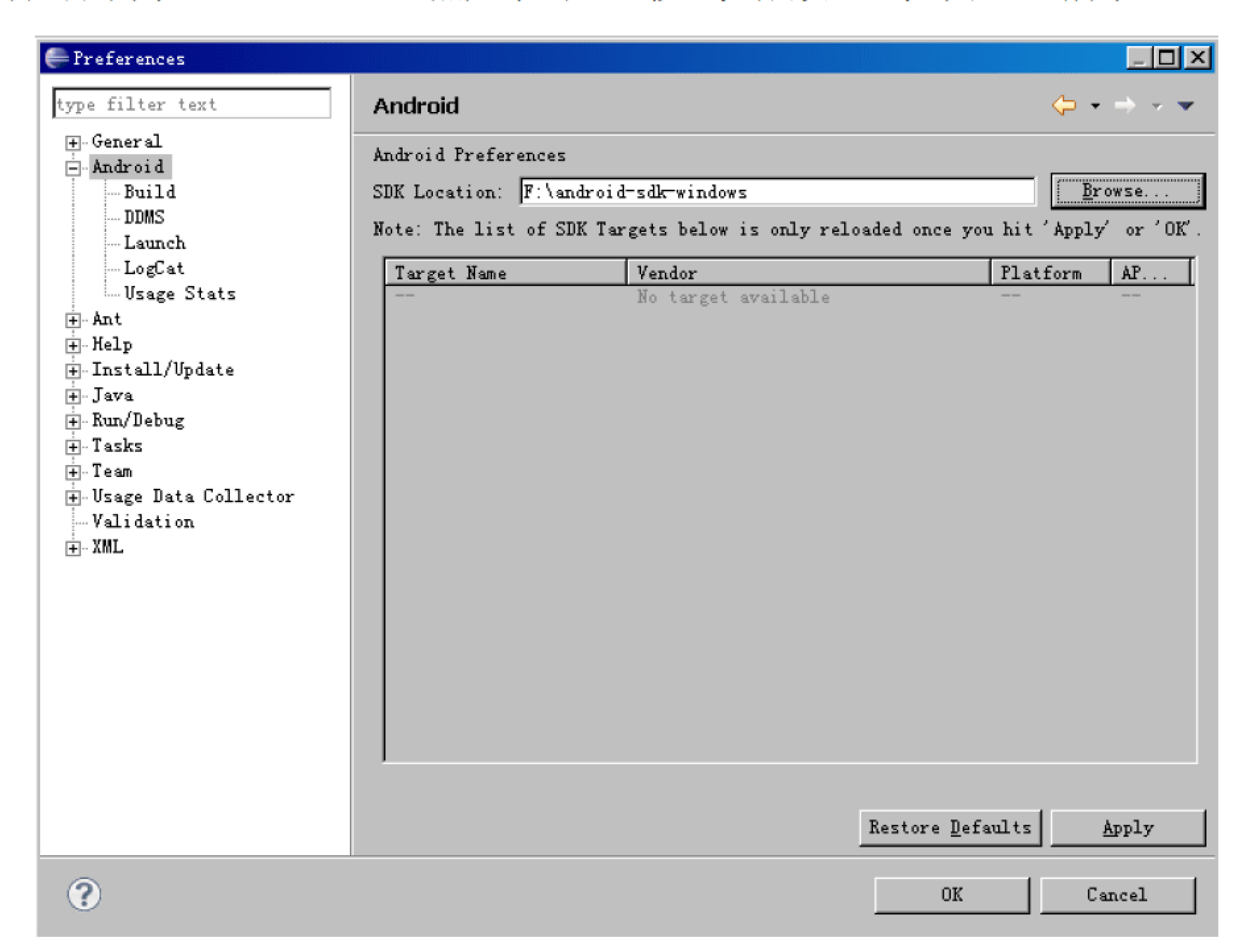


图 1-31 Preferences 对话框



1.4　创建 Android 虚拟设备(AVD)

开发的 Android 程序需要调试，只有经过调试之后才能知道我们的程序是否可以运行。作为一款手机系统，怎样在电脑上调试 Android 程序呢？谷歌为我们提供了模拟器来解决这个问题。模拟器是指在电脑上模拟 Android 系统，我们可以利用这个模拟器来调试并运行开发的 Android 程序。由此可见，模拟器的好处是，开发人员不需要一个真实的 Android 手机，仅通过电脑就可以调试并运行 Android 程序。Android 模拟器在电脑上的运行效果如图 1-32 所示。

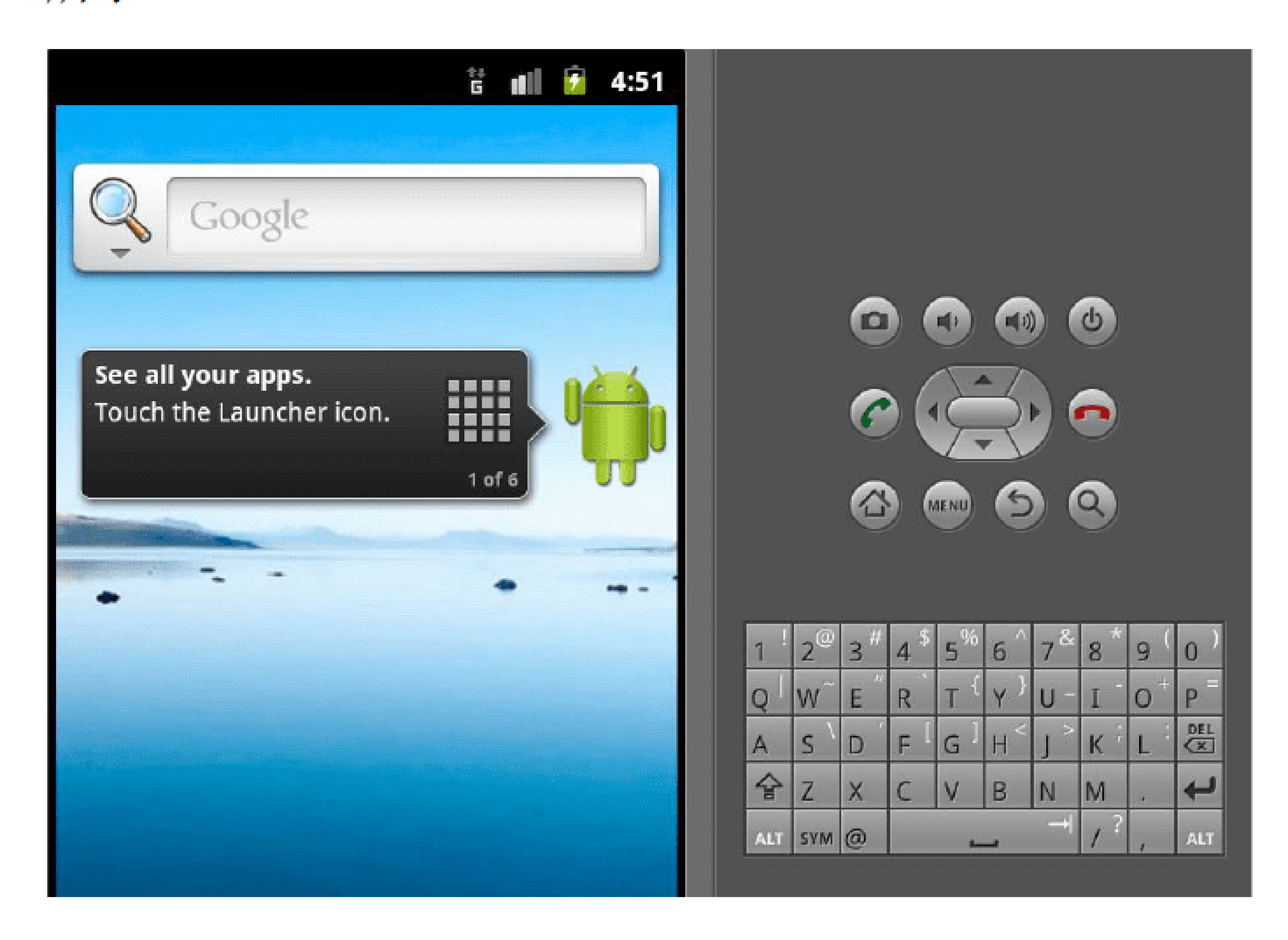


图 1-32　模拟器

1.4.1　Android 模拟器简介

Android 模拟器的全称是 Android Virtual Device，简称为 AVD。对于 Android 程序的开发者来说，模拟器的推出给开发者带来了极大的方便，无论是在开发工作上还是在测试工作上。不管在 Windows 环境下还是在 Linux 环境下，Android 模拟器都可以顺利运行。并且官方提供了 Eclipse 插件，可以将模拟器集成到 Eclipse IDE 环境。当然，也可以从命令行启动 Android 模拟器。

获取模拟器的方法非常简单，读者只需要登录 Android 官方站点(http://developer.Android.com/)即可免费下载单独的模拟器。另外，也可以先下载 Android SDK，解压后在其 SDK 的根目录下有一个名为 tools 的文件夹，此文件夹下包含了完整的模拟器和一些非常有用的工具。

Android SDK 中包含的模拟器的功能非常齐全，例如电话本、通话等功能都可以正常使用(当然不能使用模拟器实现真正的通话和短信功能)，并且其内置的浏览器和 Maps 都可以联网使用。另外，我们完全可以使用键盘输入，鼠标点击模拟器按键输入，甚至还可以使用鼠标点击、拖动屏幕进行操纵。

1.4.2 模拟器和真机的区别

当然 Android 模拟器不能完全替代真机，具体来说，模拟器和真机有如下差异。

- 模拟器不支持呼叫和接听实际来电；但可以通过控制台模拟电话呼叫(呼入和呼出)。
- 模拟器不支持 USB 连接。
- 模拟器不支持“相机/视频”捕捉。
- 模拟器不支持音频输入(捕捉)，但支持输出(重放)。
- 模拟器不支持扩展耳机。
- 模拟器不能确定连接状态。
- 模拟器不能确定电池电量水平和交流充电状态。
- 模拟器不能确定 SD 卡的插入/弹出。
- 模拟器不支持蓝牙。

1.4.3 创建 Android 虚拟设备

每个 AVD 模拟了一套虚拟设备来运行 Android 系统，在每个 AVD 中至少要有自己的内核、系统图像和数据分区，还可以有自己的 SD 卡和用户数据以及外观显示等。创建 AVD 的基本步骤如下。

(1) 单击 Eclipse 菜单中的按钮，如图 1-33 所示。

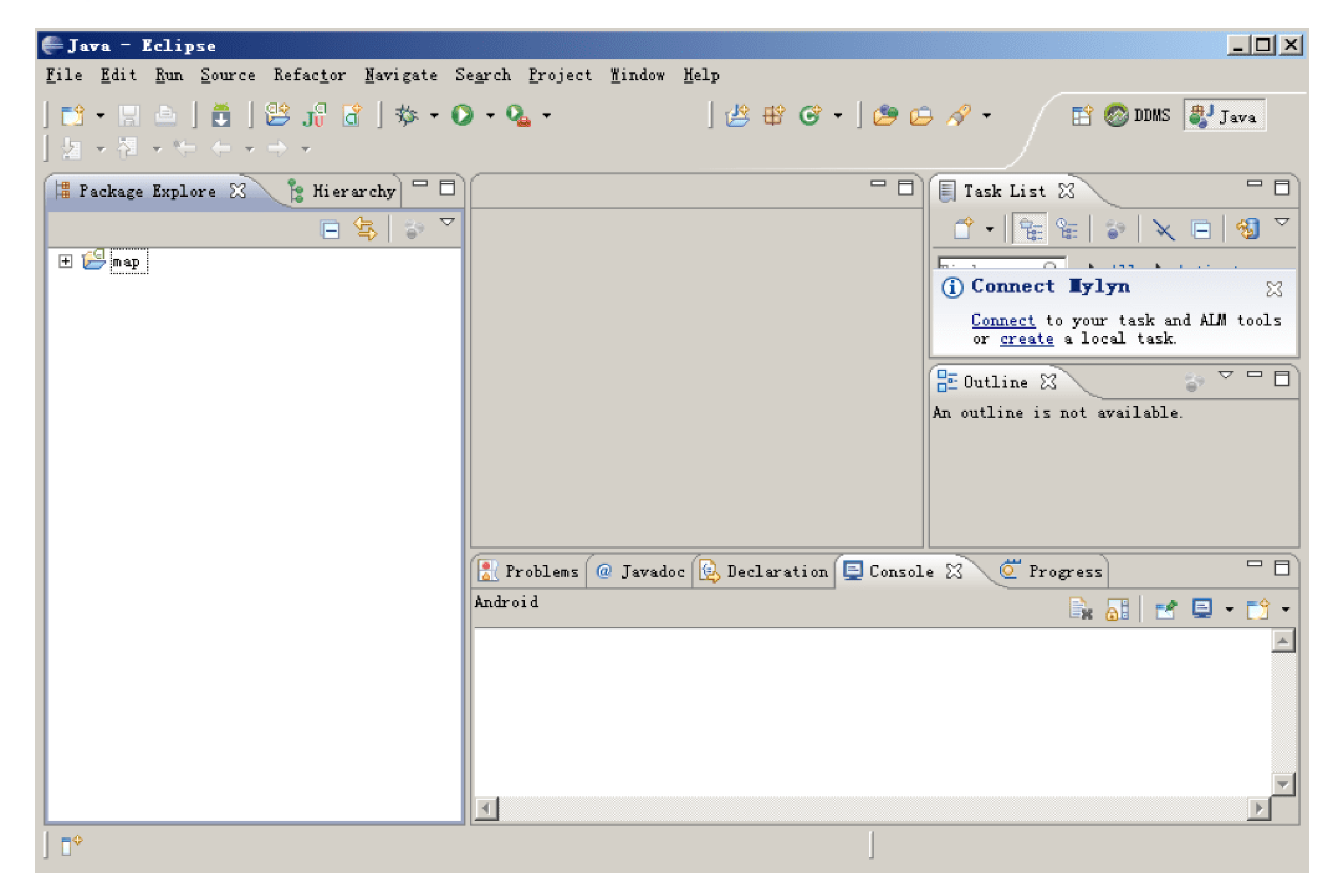


图 1-33 Eclipse 窗口

(2) 弹出 Android SDK and AVD Manager 对话框，在左侧导航栏中选择 Virtual devices 选项，如图 1-34 所示。

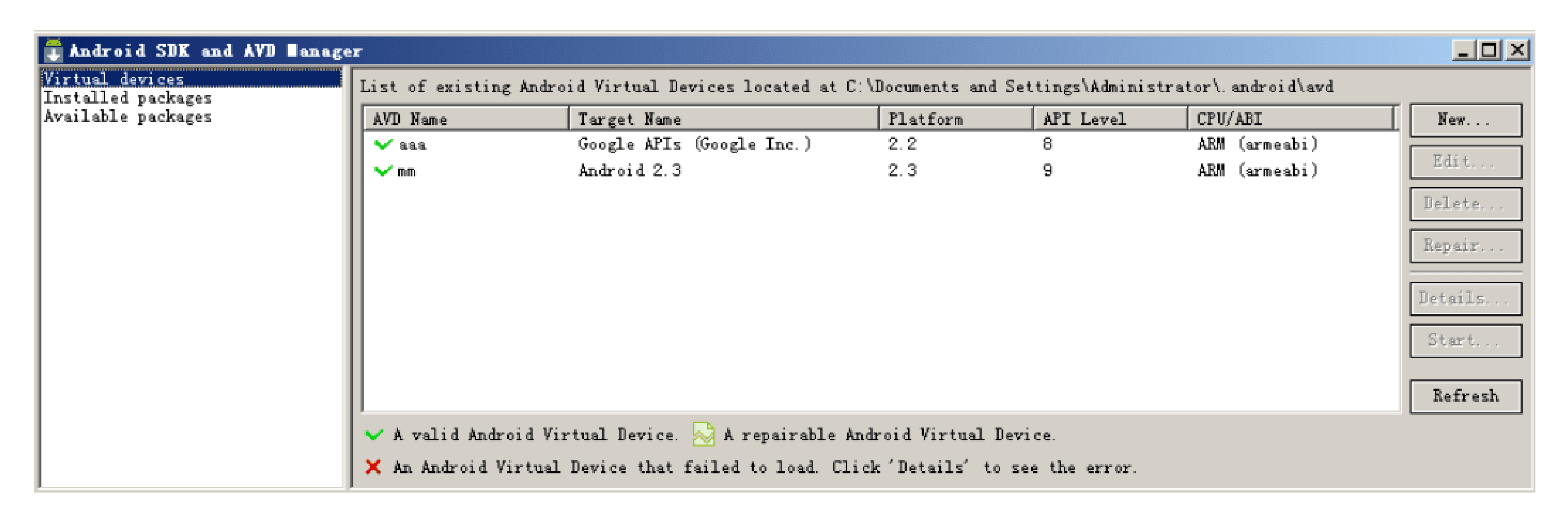


图 1-34　Android SDK and AVD Manager 对话框

在 Virtual device 列表中列出了当前已经安装的 AVD 版本，可以通过右侧的按钮来创建、删除或修改 AVD。主要按钮的具体说明如下。

- New：创建新的 AVD，单击此按钮在弹出的对话框中可以创建一个新的 AVD，如图 1-35 所示。
- Edit：修改已经存在的 AVD。
- Delete：删除已经存在的 AVD。
- Start：启动一个 AVD 模拟器。

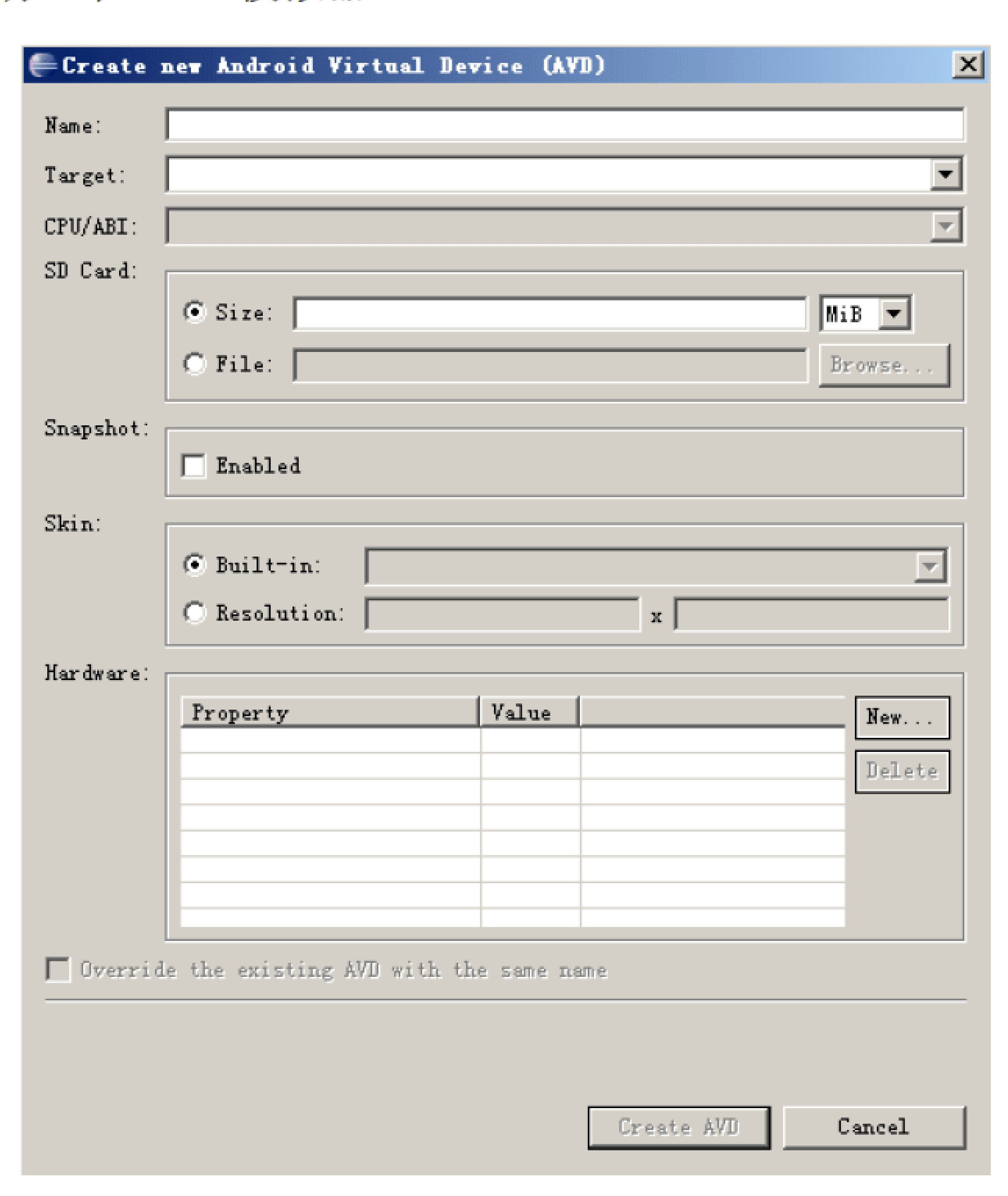


图 1-35　新建 AVD 对话框

除了上面介绍的创建 AVD 的方法之外，也可以在 CMD 窗口中创建或删除 AVD，例如可以用如下 CMD 命令创建一个 AVD。

```
android create avd --name <your_avd_name> --target <targetID>
```

其中“your_avd_name”是需要创建的 AVD 的名字，CMD 窗口如图 1-36 所示。

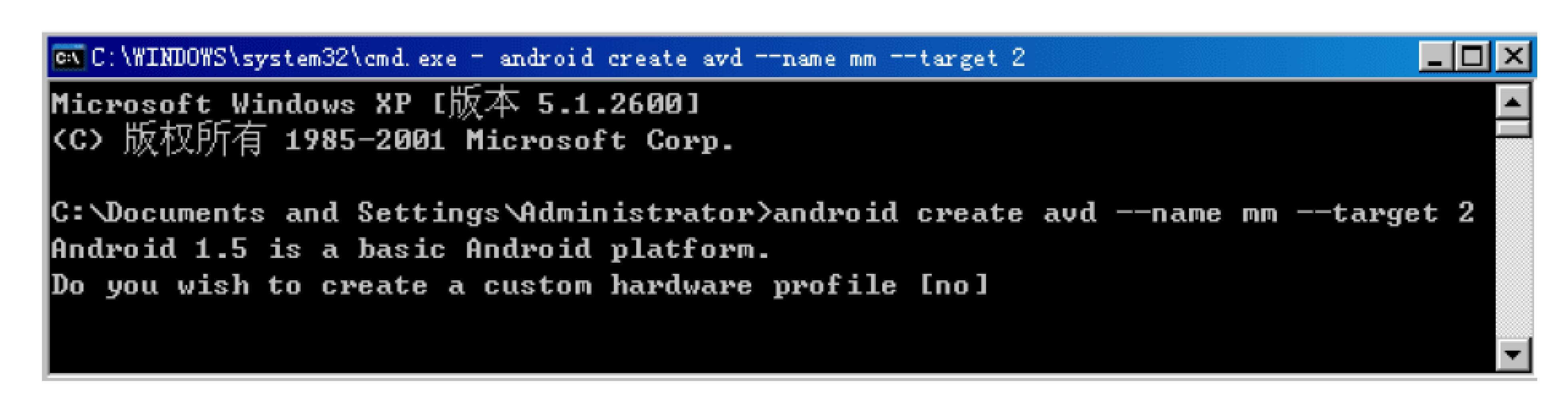


图 1-36　CMD 窗口

1.4.4　启动模拟器

在调试的时候需要先启动 AVD 模拟器。启动 AVD 模拟器的基本流程如下。

(1) 选择图 1-34 列表中名为 mm 的 AVD，单击 Start 按钮后弹出 Launch Option 对话框，如图 1-37 所示。

(2) 单击 Launch 按钮后将会运行名为 mm 的模拟器，如图 1-38 所示。

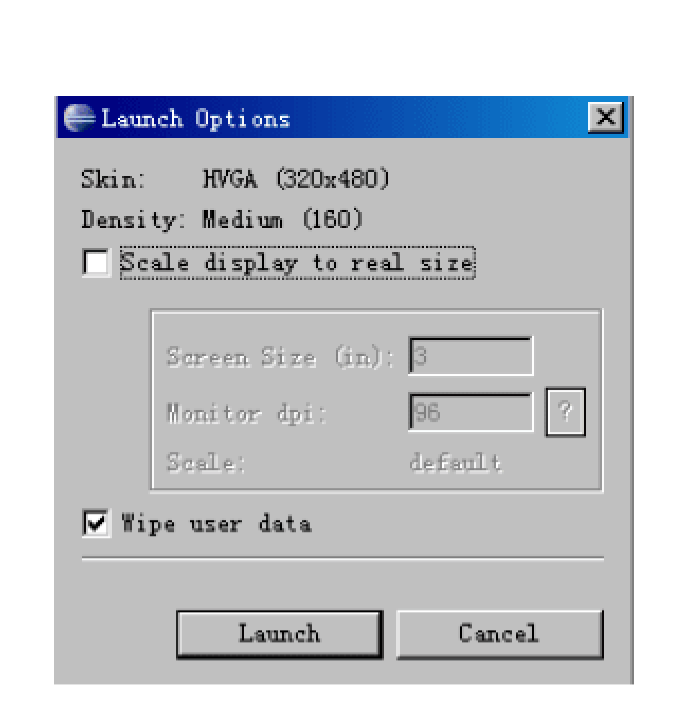


图 1-37　Launch Options 对话框

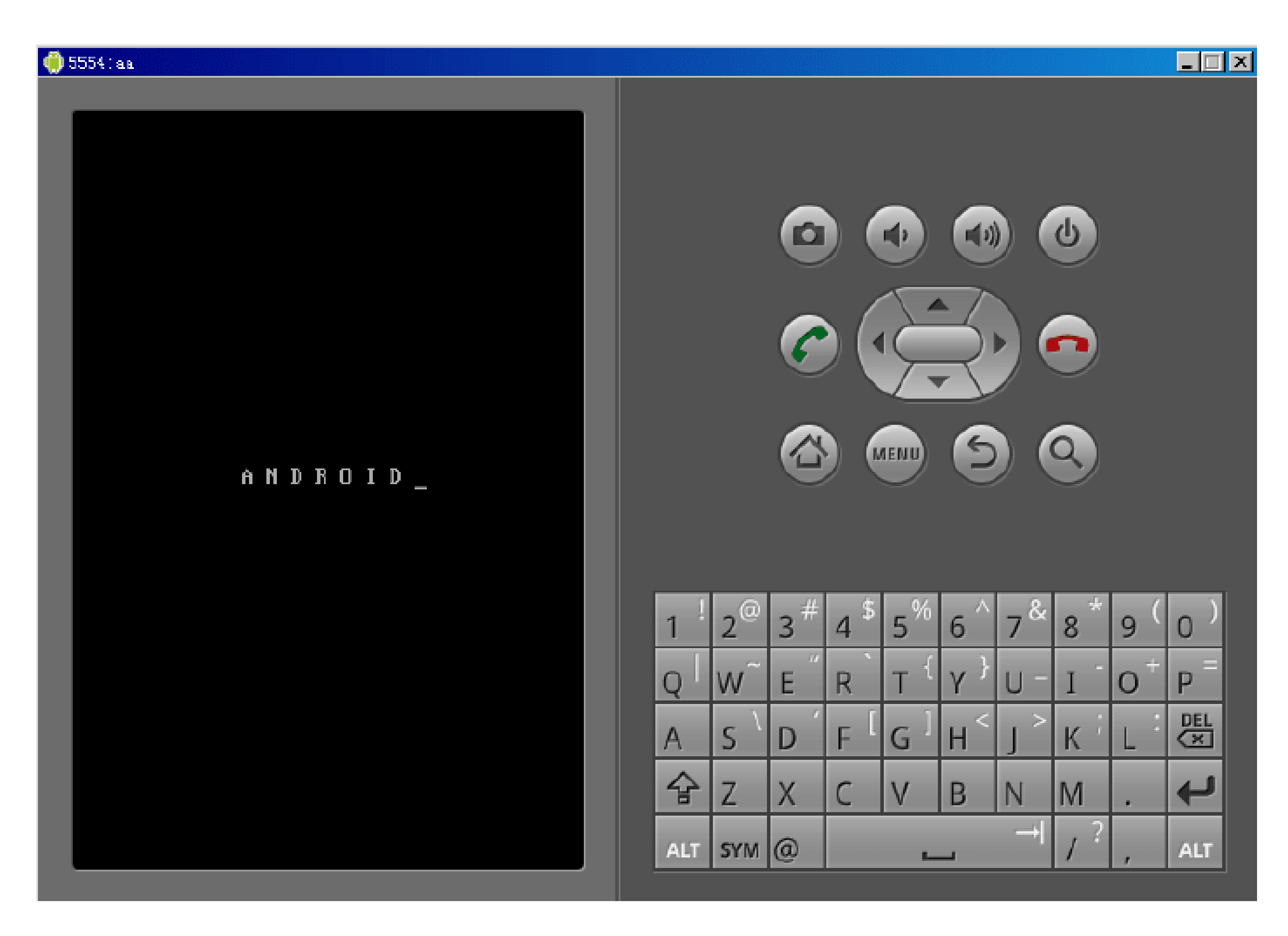
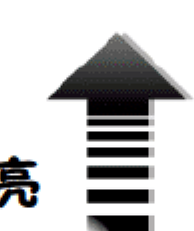


图 1-38　模拟运行成功

1.4.5　快速安装 SDK

通过 Android SDK Manager 在线安装的速度非常慢，而且有时容易掉线。其实我们可以先从网络中寻找到 SDK 资源，用迅雷等下载工具下载后，将其放到指定目录后就可以完成安装。具体方法是先下载 android-sdk-windows(可以更新的那种)，然后在 android-sdk-windows 下双击 setup.exe，在更新的过程中会发现安装 Android SDK 的速度是 1kb/s，此时



打开迅雷，分别输入下面的地址：

https://dl-ssl.google.com/android/repository/platform-tools_r05-windows.zip
https://dl-ssl.google.com/android/repository/docs-3.1_r01-linux.zip
https://dl-ssl.google.com/android/repository/android-2.2_r02-windows.zip
https://dl-ssl.google.com/android/repository/android-2.3.3_r01-linux.zip
https://dl-ssl.google.com/android/repository/android-2.1_r02-windows.zip
https://dl-ssl.google.com/android/repository/samples-2.3.3_r01-linux.zip
https://dl-ssl.google.com/android/repository/samples-2.2_r01-linux.zip
https://dl-ssl.google.com/android/repository/samples-2.1_r01-linux.zip
https://dl-ssl.google.com/android/repository/compatibility_r02.zip
https://dl-ssl.google.com/android/repository/tools_r11-windows.zip
https://dl-ssl.google.com/android/repository/google_apis-10_r02.zip
https://dl-ssl.google.com/android/repository/android-2.3.1_r02-linux.zip
https://dl-ssl.google.com/android/repository/usb_driver_r04-windows.zip
https://dl-ssl.google.com/android/repository/googleadmobadssdkandroid-4.1.0.zip
https://dl-ssl.google.com/android/repository/market_licensing-r01.zip
https://dl-ssl.google.com/android/repository/market_billing_r01.zip
https://dl-ssl.google.com/android/repository/google_apis-8_r02.zip
https://dl-ssl.google.com/android/repository/google_apis-7_r01.zip
https://dl-ssl.google.com/android/repository/google_apis-9_r02.zip
…

可以继续根据自己开发要求选择不同版本的 API。

下载完后将它们复制到 android-sdk-windows/Temp 目录下，然后再运行 setup.exe，选中需要的 API 选项，会发现马上就安装好了。记得把原始文件保留好，因为放在 Temp 目录下的文件安装好后会立即消失。

1.5　解决搭建环境过程中的三个问题

本节将总结在搭建 Android SDK 环境过程中常见的三个问题，帮助读者快速成功搭建 Android 开发环境。

1.5.1　不能在线更新

在安装 Android SDK 后，需要及时更新为最新的资源和配置。但是在启动 Android 后，经常会发生不能正常更新的情况，例如会弹出如图 1-39 所示的错误提示。

Android 默认的在线更新地址是 https://dl-ssl.google.com/android/eclipse/，但是经常会出现错误。如果此地址不能更新，可以自行设置更新地址，修改为 http://dl-ssl.google.com/android/repository/repository.xml。具体操作方法如下。



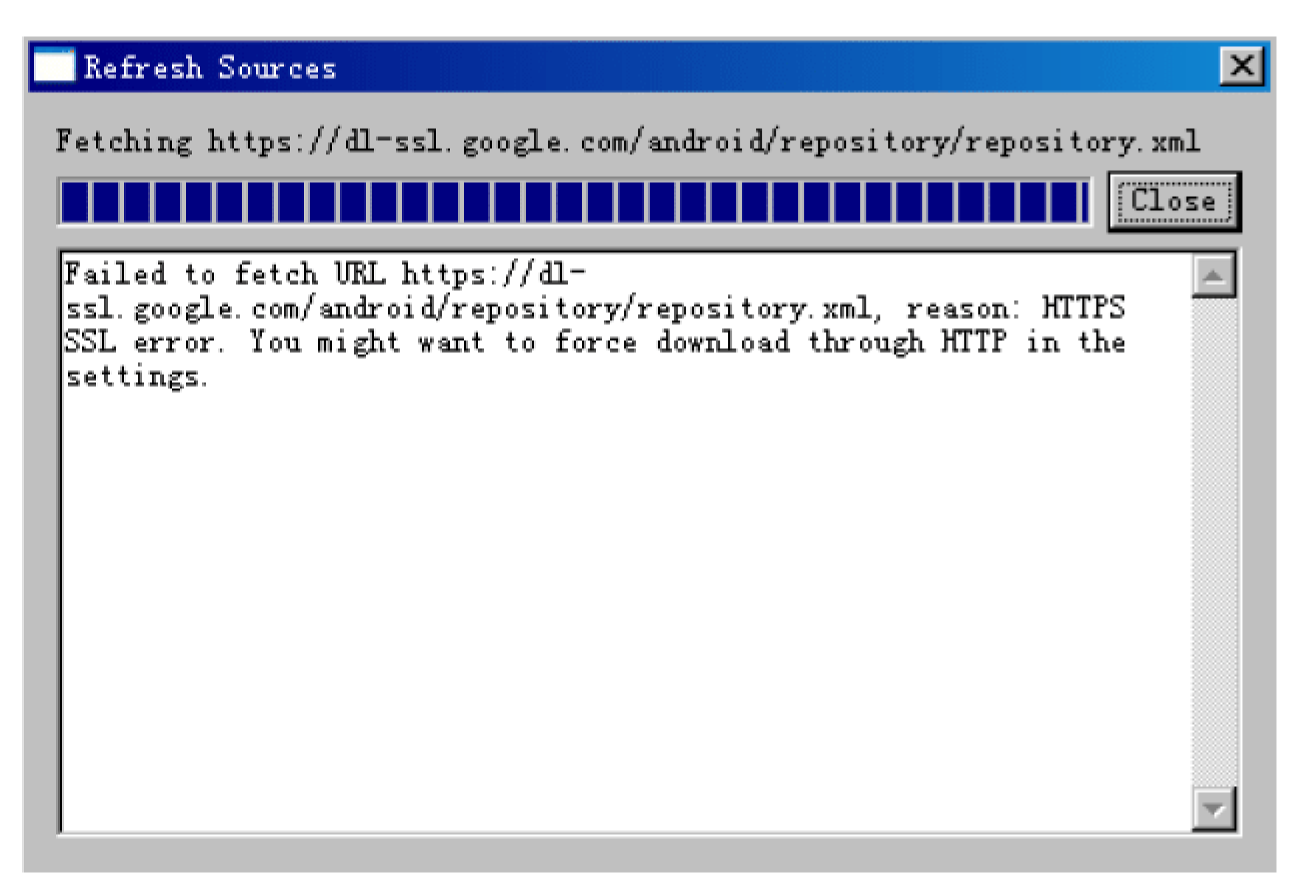


图 1-39　不能更新

(1) 单击 Android SDK and AVD Manager 对话框左侧的 Available Packages 选项，然后单击下面的 Add Site 按钮，如图 1-40 所示。

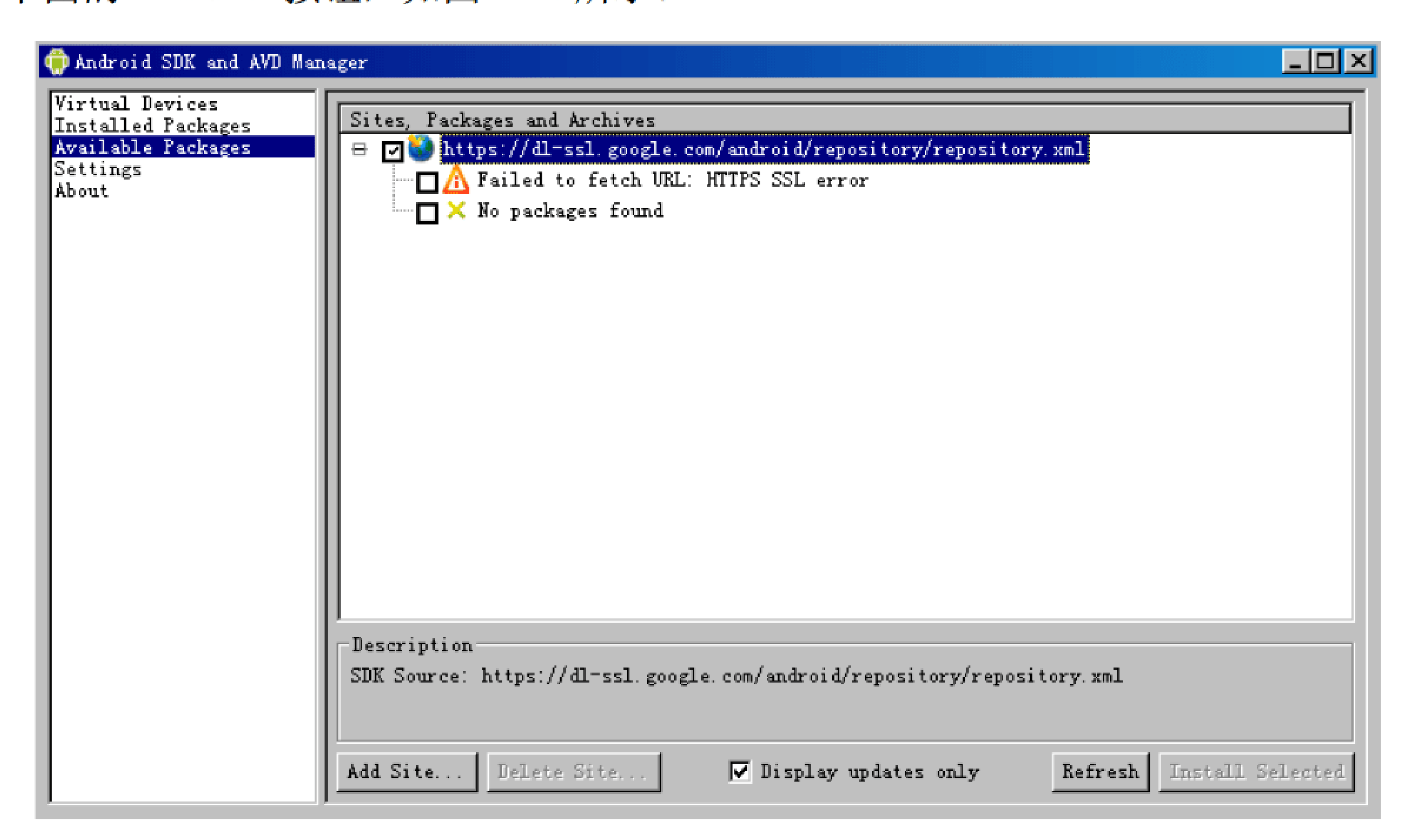


图 1-40　Available Packages 界面

(2) 在弹出的 Add Site URL 对话框中输入如下修改后的地址，如图 1-41 所示。

```
http://dl-ssl.google.com/android/repository/repository.xml
```

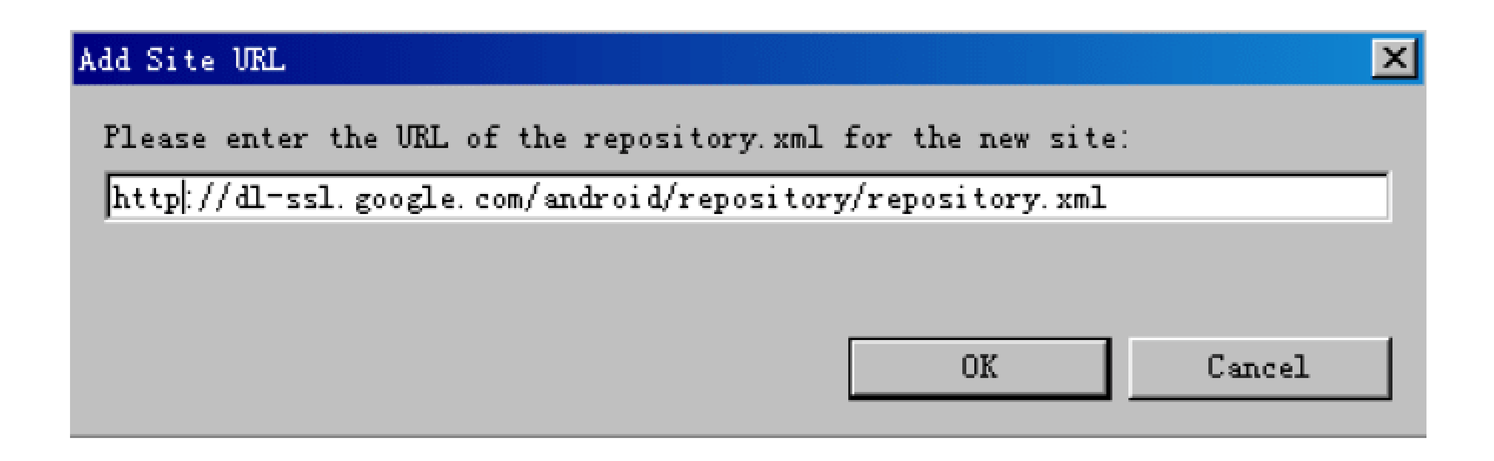


图 1-41　Add Site URL 对话框

(3) 单击 OK 按钮完成设置，此时就可以使用更新功能了，如图 1-42 所示。

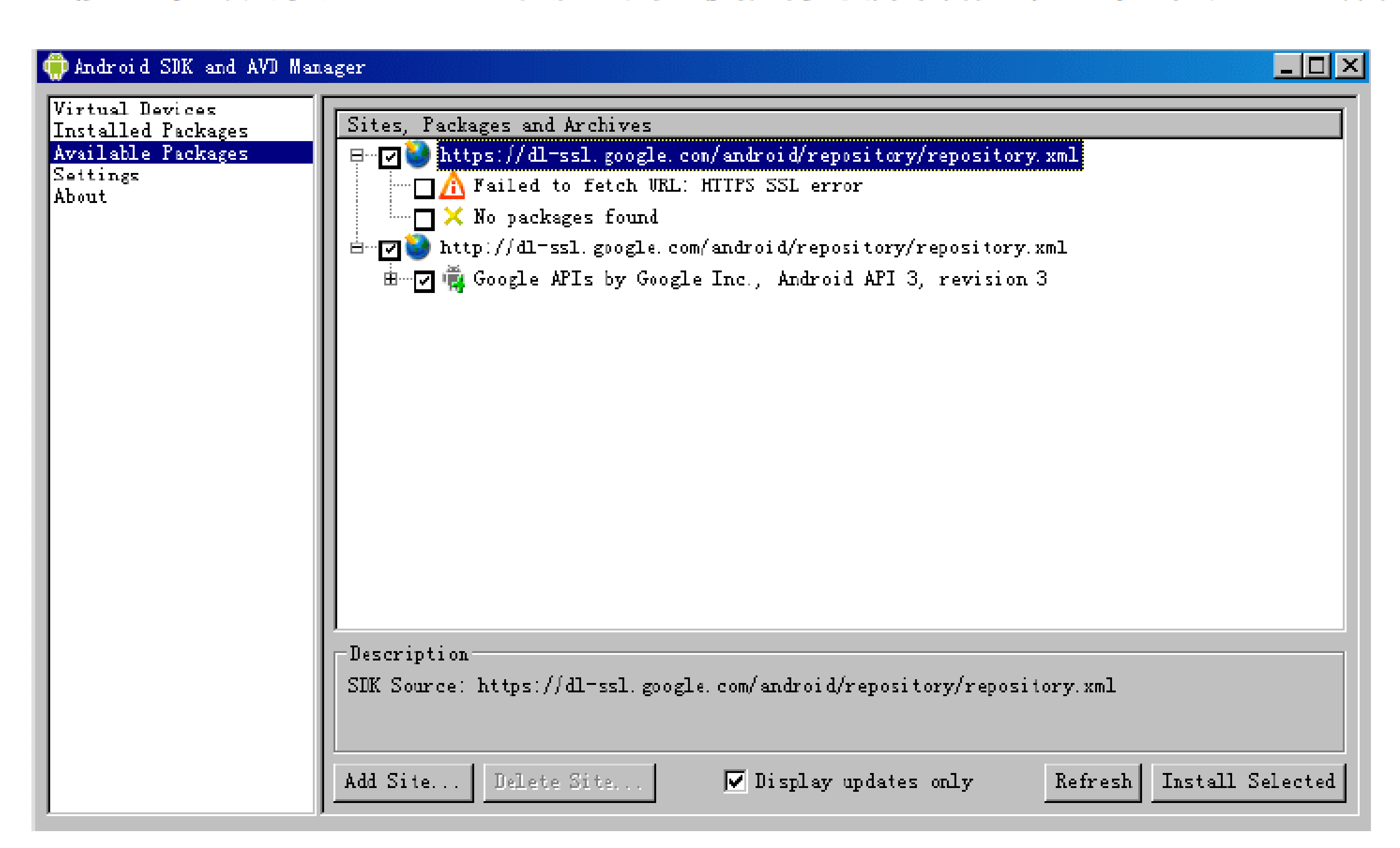


图 1-42　Available Packages 界面

1.5.2　一直显示 Project name must be specified 提示

在 Eclipse 中新建 Android 工程时，一直提示“Project name must be specified”，如图 1-43 所示。

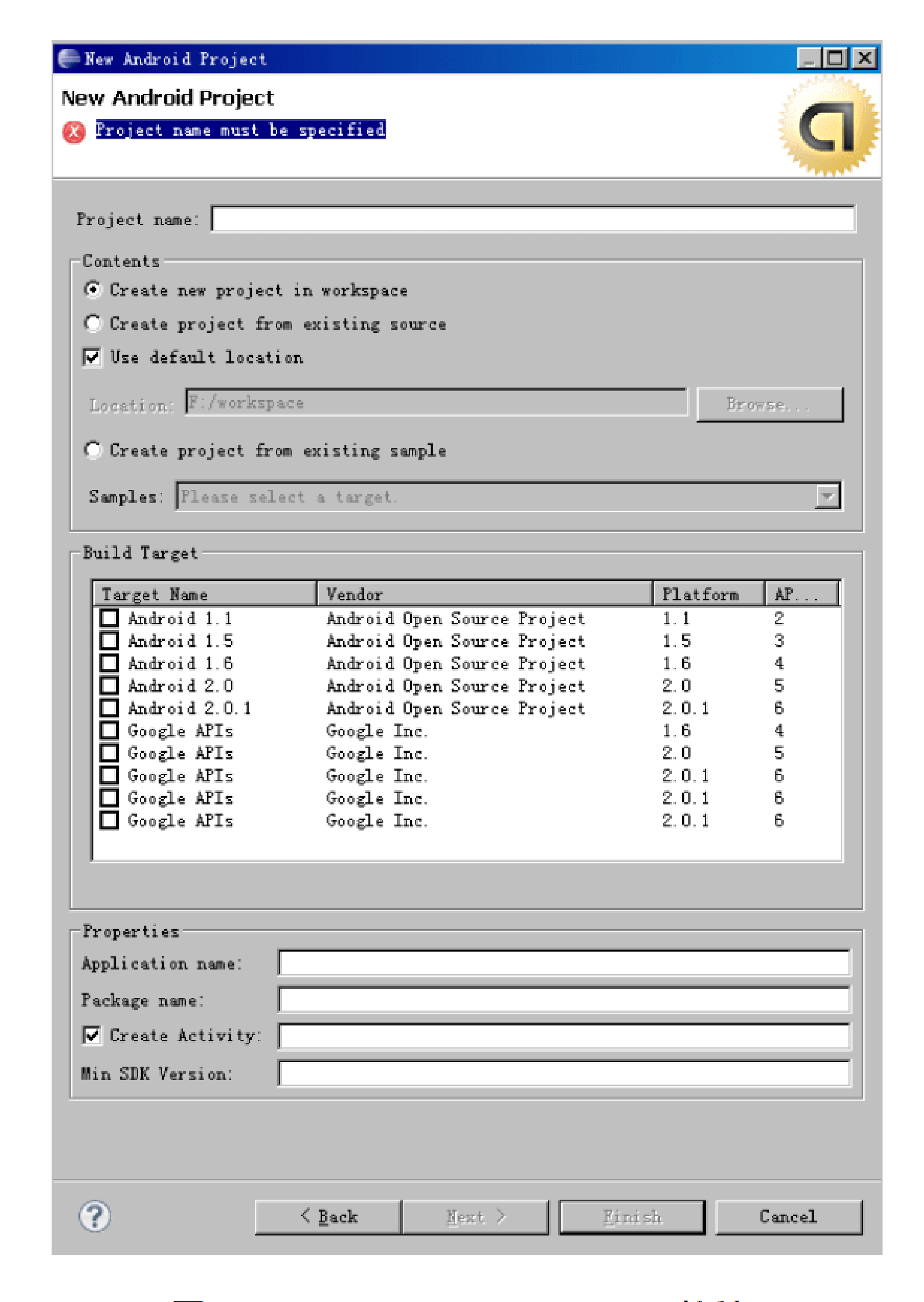


图 1-43　New Android Project 对话框

造成上述问题的原因是 Android 没有更新完成，需要进行完全更新，具体方法如下。

(1) 打开 Android SDK and AVD Manager 对话框，选择左侧的 Installed Packages 选项，如图 1-44 所示。

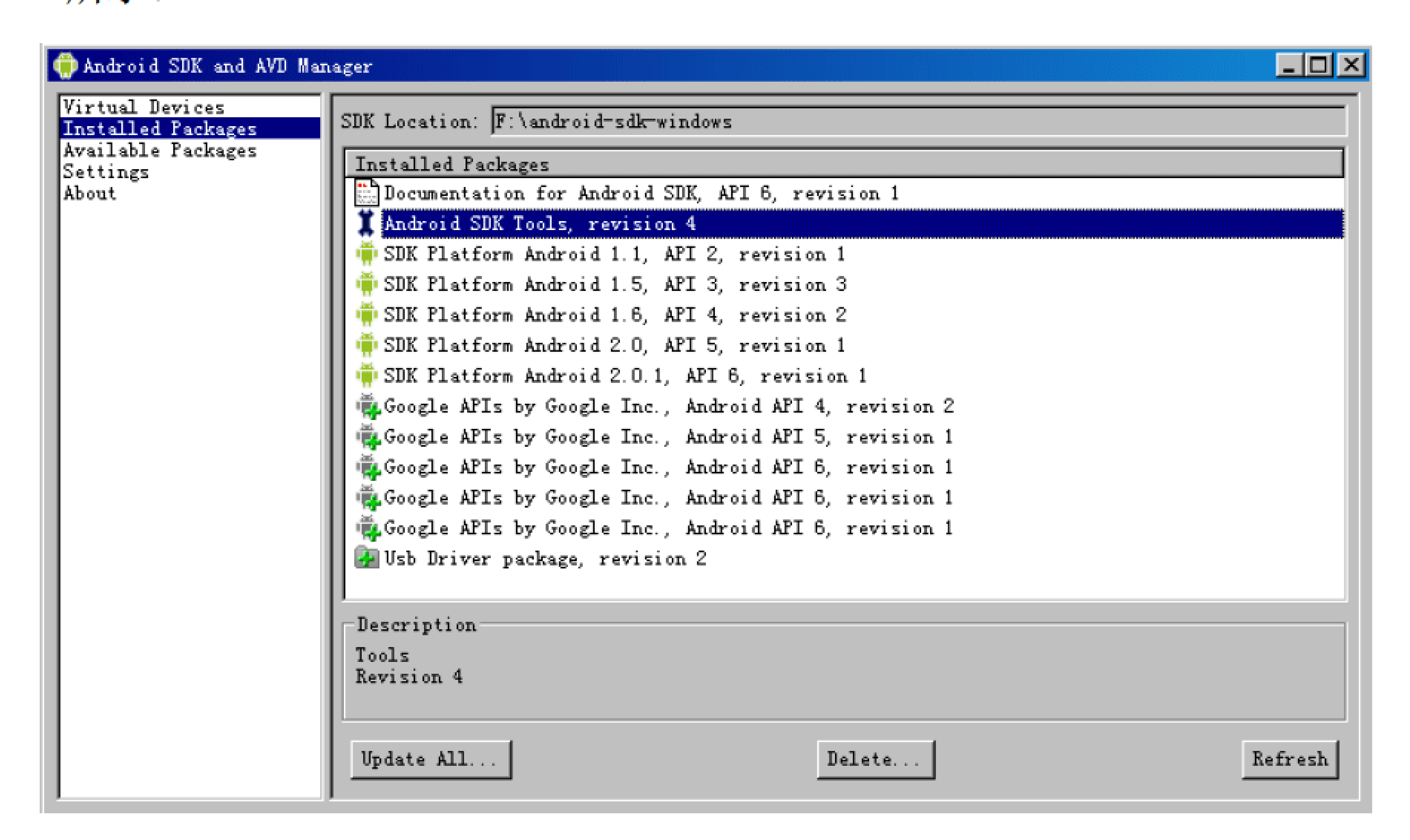


图 1-44　Installed Packages 界面

(2) 在右侧列表中选择“Android SDK Tools, revision 4”选项，在弹出的对话框中选中 Accept 单选按钮，最后单击 Install Accepted 按钮开始安装更新，如图 1-45 所示。

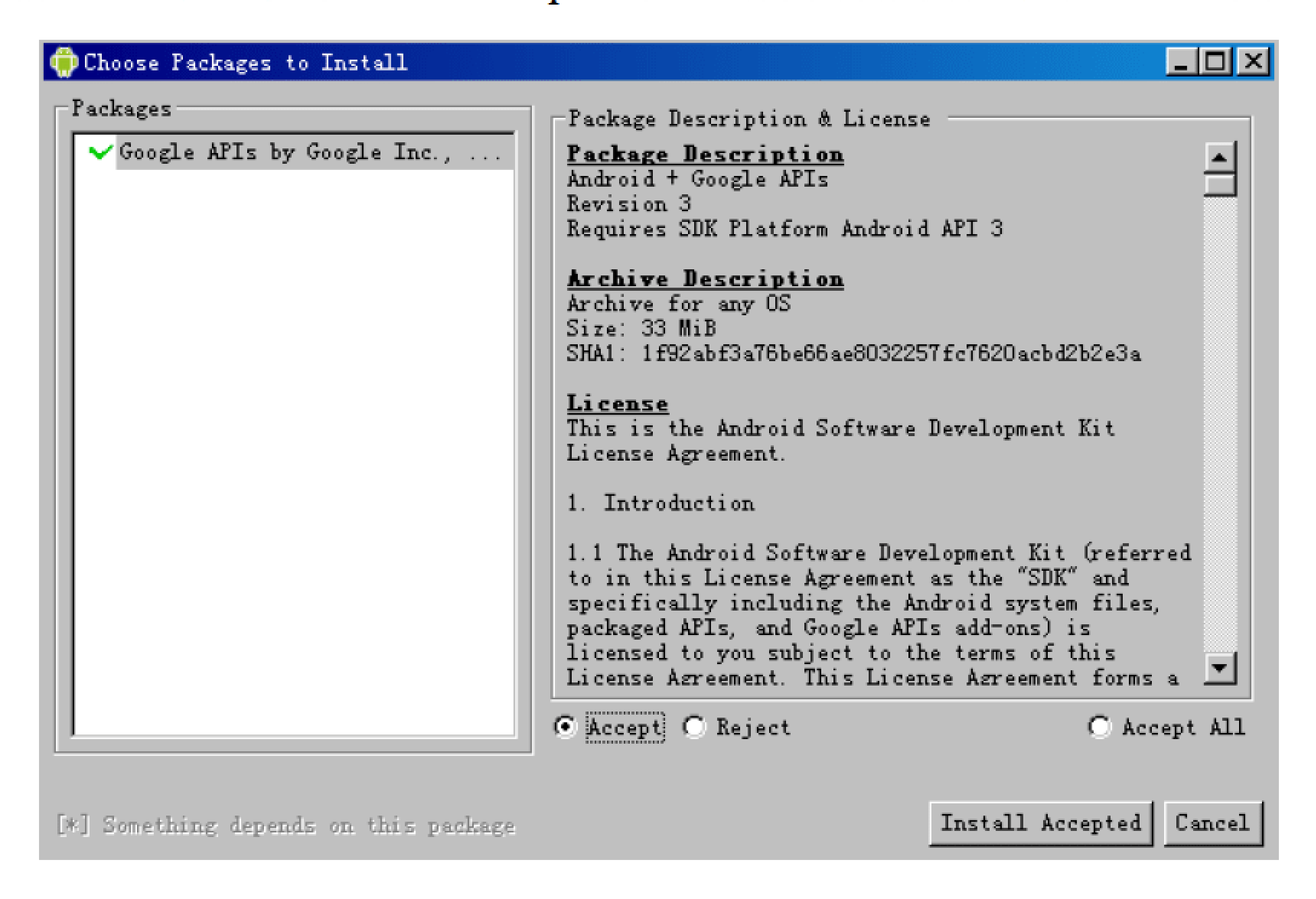
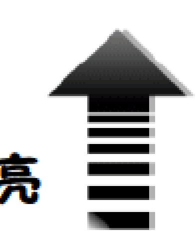


图 1-45　Choose Packages to Install 对话框

1.5.3　Target 列表中没有 Target 选项

当搭建 Android 开发环境完毕后，在 Eclipse 菜单栏中依次选择 Window | Preference 命令，然后在弹出的 Preferences 对话框中单击左侧的 Android 选项，会在 Preferences 中显



示当前系统已经存在的 SDK Targets，如图 1-46 所示。

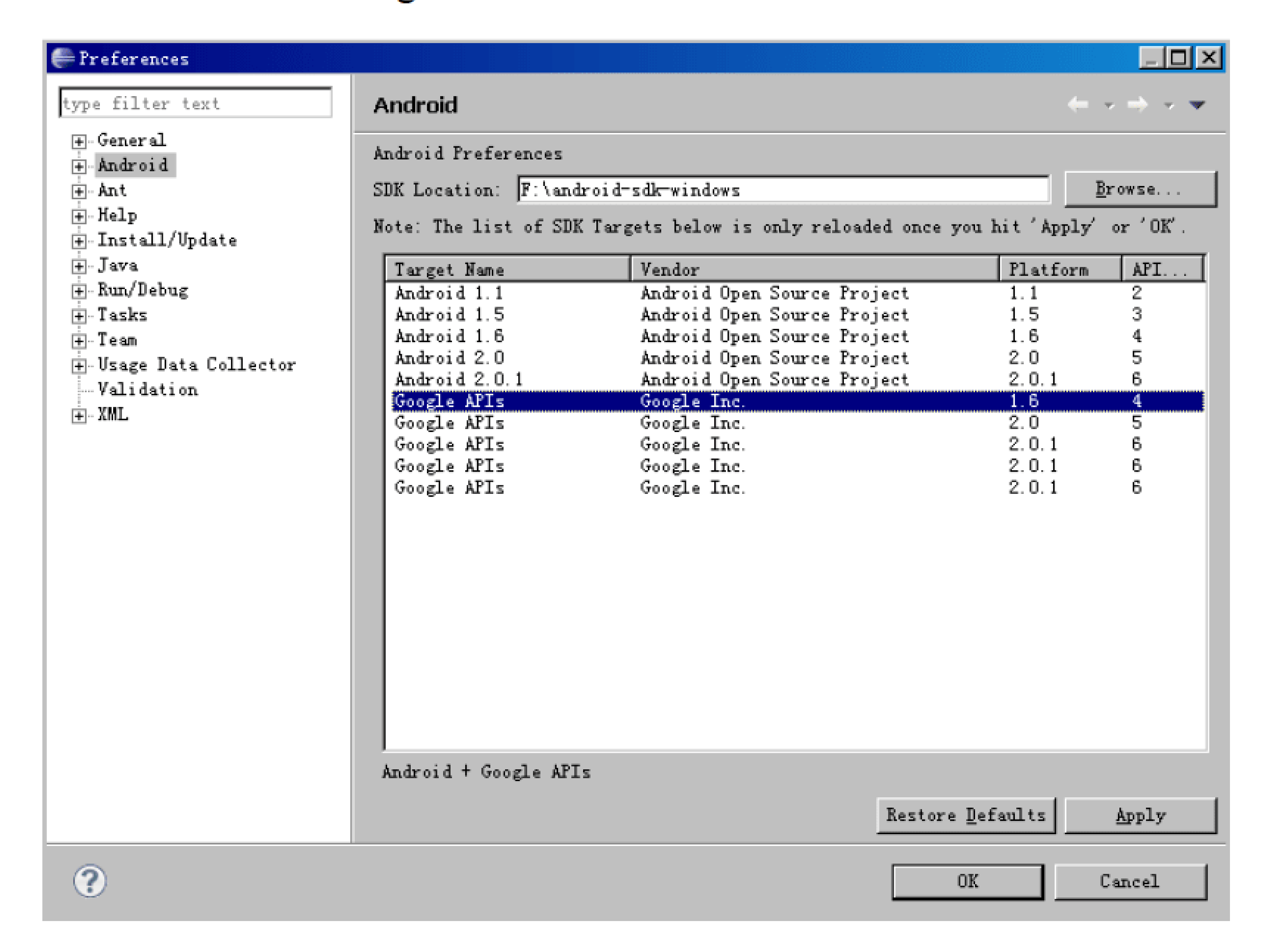


图 1-46　SDK Targets 列表

但是往往会因为各种原因不显示 SDK Targets 列表，并且在图 1-31 所示的对话框中也不显示 SDK Targets 列表，并输出“Failed to find an AVD compatible with target”的错误提示。

造成上述问题的原因是没有成功创建 AVD，此时需要我们手工安装来解决这个问题，当然前提是 Android 已更新完毕。具体解决方法如下。

(1) 在【运行】对话框中输入“CMD”，打开 CMD 窗口，如图 1-47 所示。

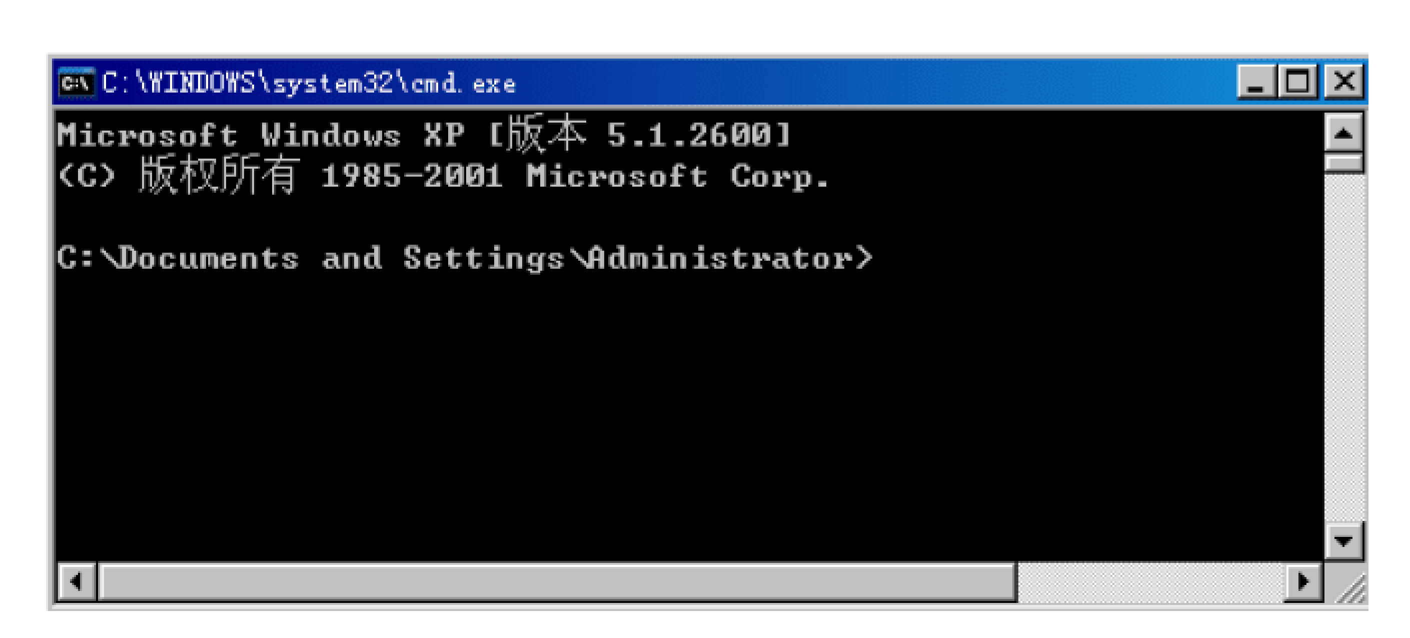


图 1-47　CMD 窗口

(2) 使用如下 Android 命令创建一个 AVD。

```
android create avd --name <your_avd_name> --target <targetID>
```

其中“your_avd_name”表示需要创建的 AVD 的名字，此时 CMD 窗口如图 1-48 所示。

图 1-48 的窗口中创建了一个名为 aa，target ID 为 3 的 AVD，然后在 CMD 窗口中输入“n”，即完成创建，如图 1-49 所示。

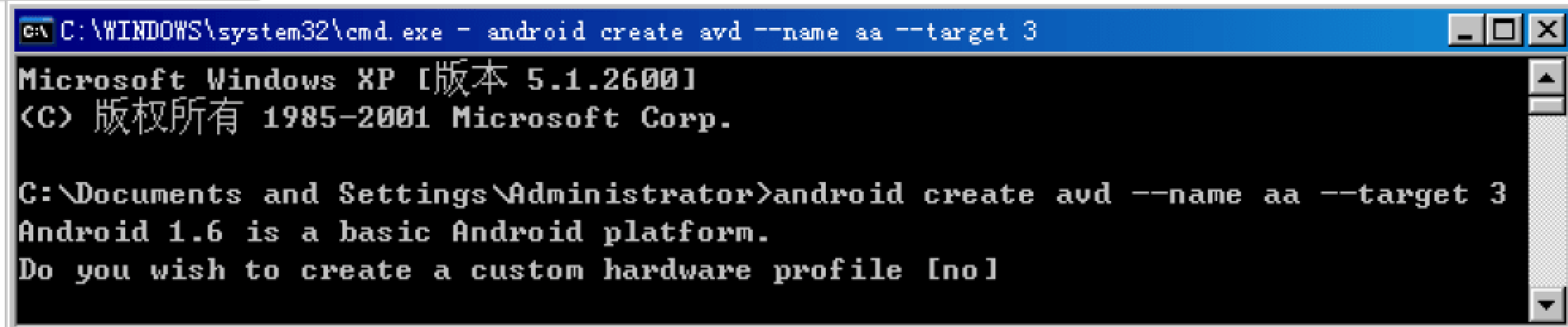


图 1-48　CMD 窗口

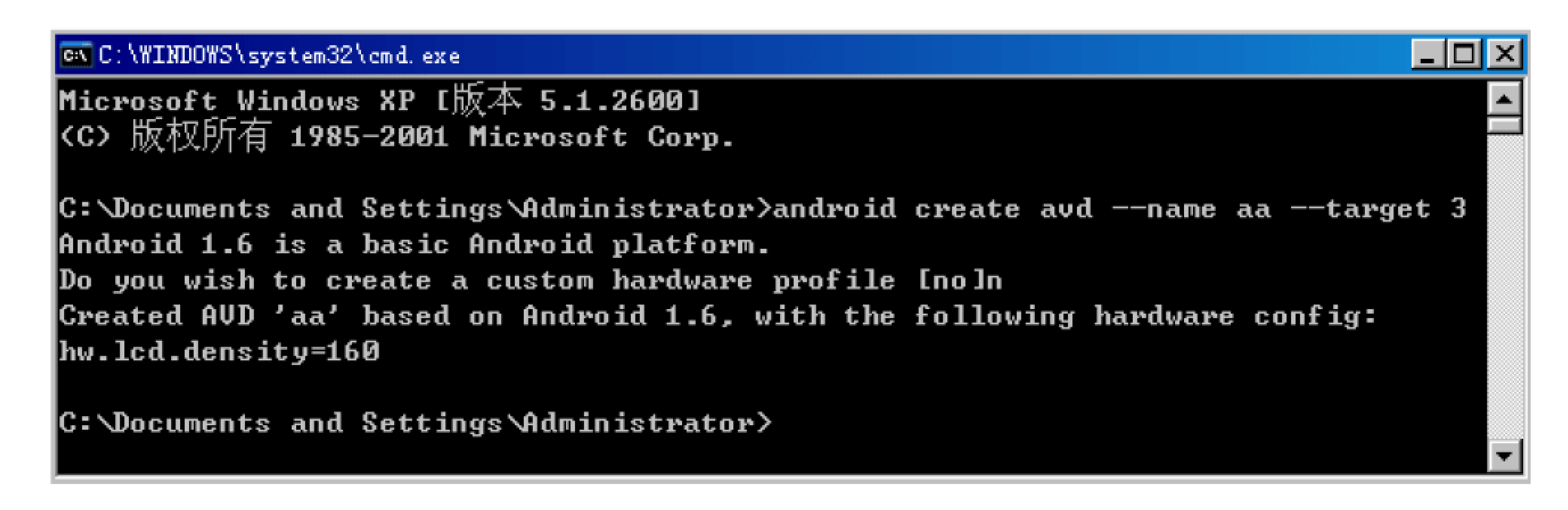


图 1-49　创建完成

Android

第2章

分析 Android 核心框架

学习 Android 开发技术需要掌握很多知识点，例如底层接口、网络应用、多媒体应用、游戏应用和优化技术等，而本书将详细讲解 Android 优化技术的基本知识。在学习这些知识之前，读者需要先了解一些基础性的知识。从本章开始将简要讲解开发 Android 游戏项目前的准备工作，为读者步入本书后面高级知识的学习打下基础。

2.1 简析 Android 安装文件

当我们下载并安装 Android SDK 后，会在安装目录中看到一些安装文件。这些文件具体有什么作用呢？在本节的内容中，将一一为大家解开这个谜题。

2.1.1 Android SDK 目录结构

安装 Android SDK 后，其安装目录的结构如图 2-1 所示。



图 2-1　Android SDK 安装后的目录结构

- add-ons：里面包含了 Android 提供的 API 包，最为主要的是和 Map 地图相关的 API 包。
- docs：里面包含了帮助文档和说明文档等常用的文档。
- platforms：针对每个版本的 SDK 版本提供了和其对应的 API 包，以及一些示例文件，其中包含了各个版本的 Android，如图 2-2 所示。

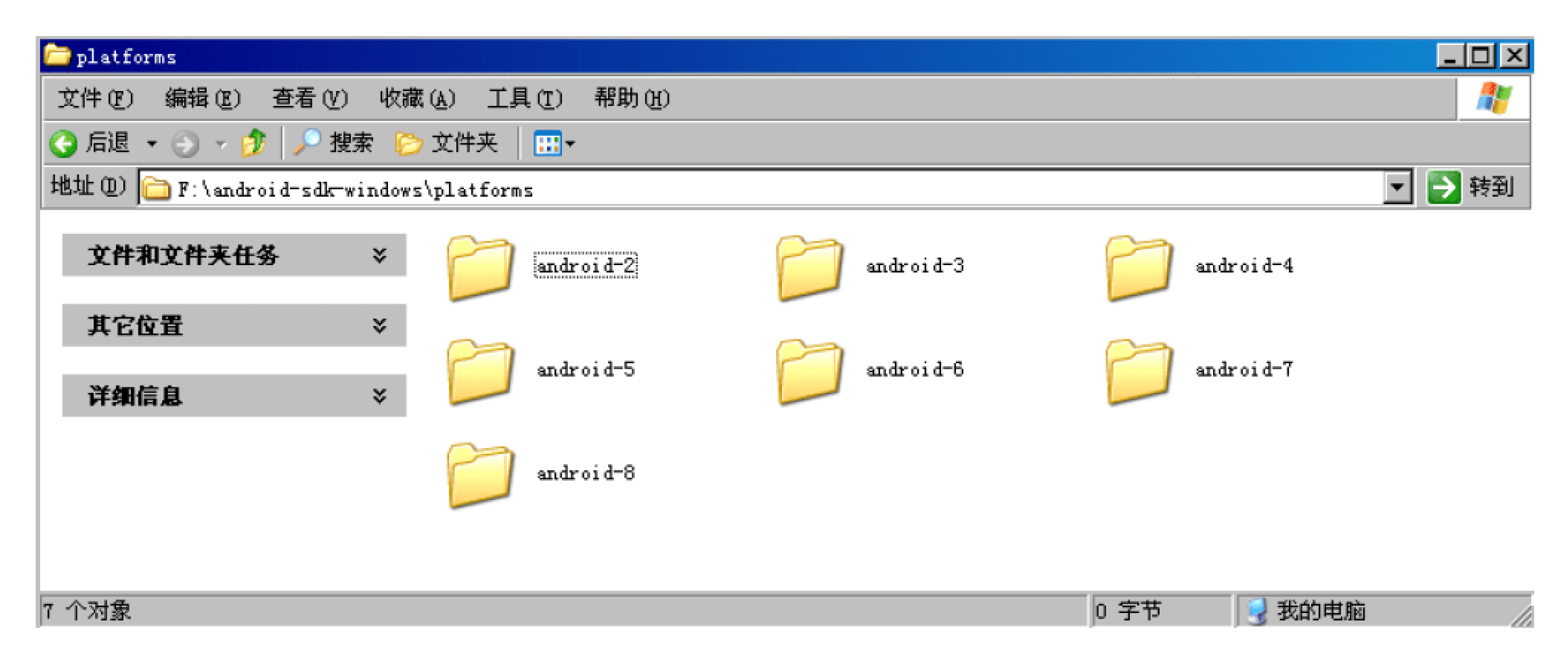


图 2-2　platforms 目录项

- temp：里面包含了一些常用的文件模板。
- tools：包含了一些通用的工具文件。
- usb_driver：包含了 AMD 64 和 x86 下的驱动文件。
- SDK Setup.exe：Android 的启动文件。

2.1.2　android.jar 及其内部结构

在 platforms 目录下的每个 Android 版本中，都有一个名为 android.jar 的文件，例如 platforms\android-8 中的 android.jar 文件如图 2-3 所示。

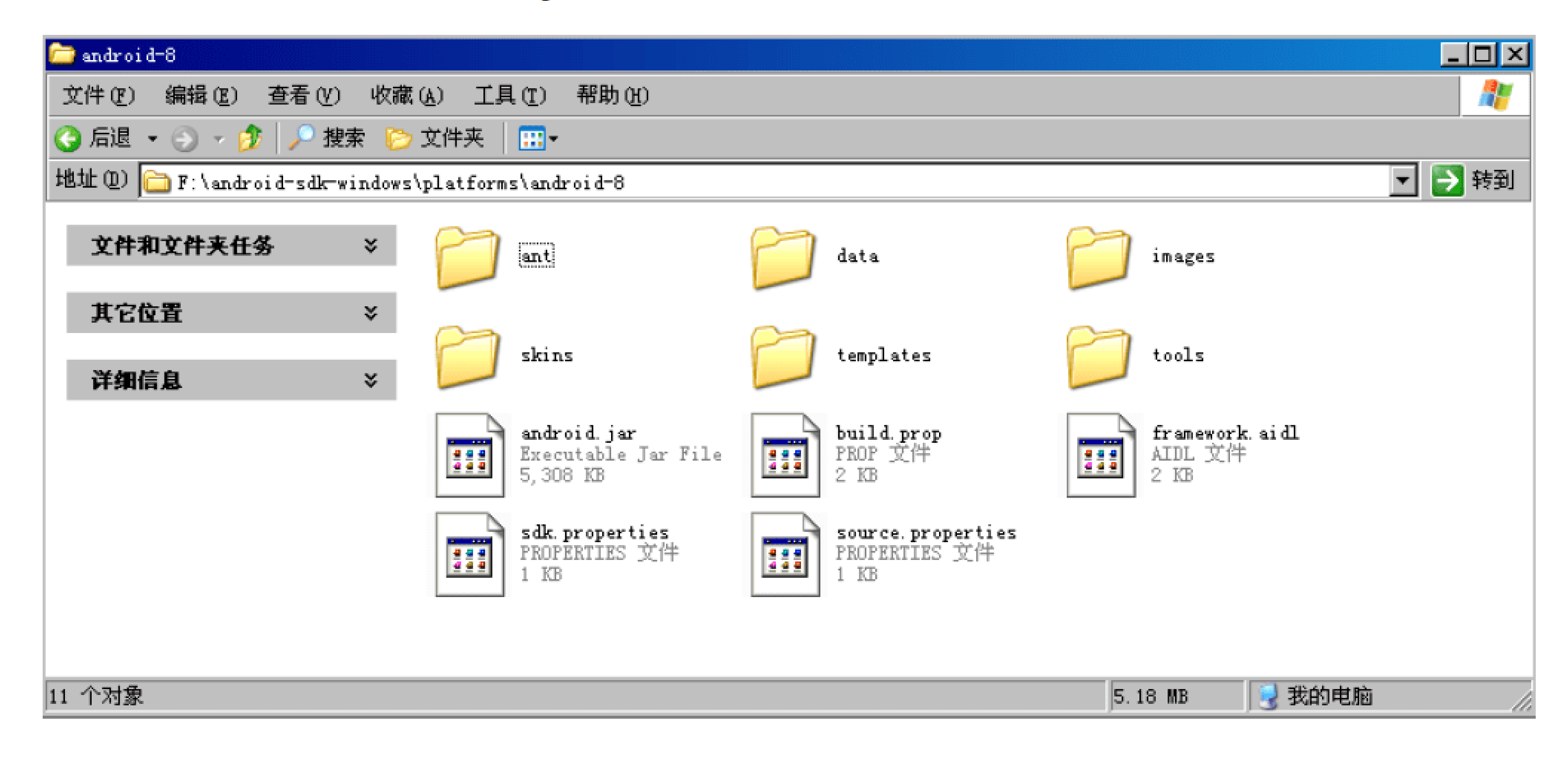


图 2-3　android.jar 文件所在目录

文件 android.jar 是一个标准的压缩包，里面包含了编译后的压缩文件和全部的 API。使用解压缩工具可以打开此压缩文件，解压后可以看到其内部结构分别如图 2-4 和图 2-5 所示。

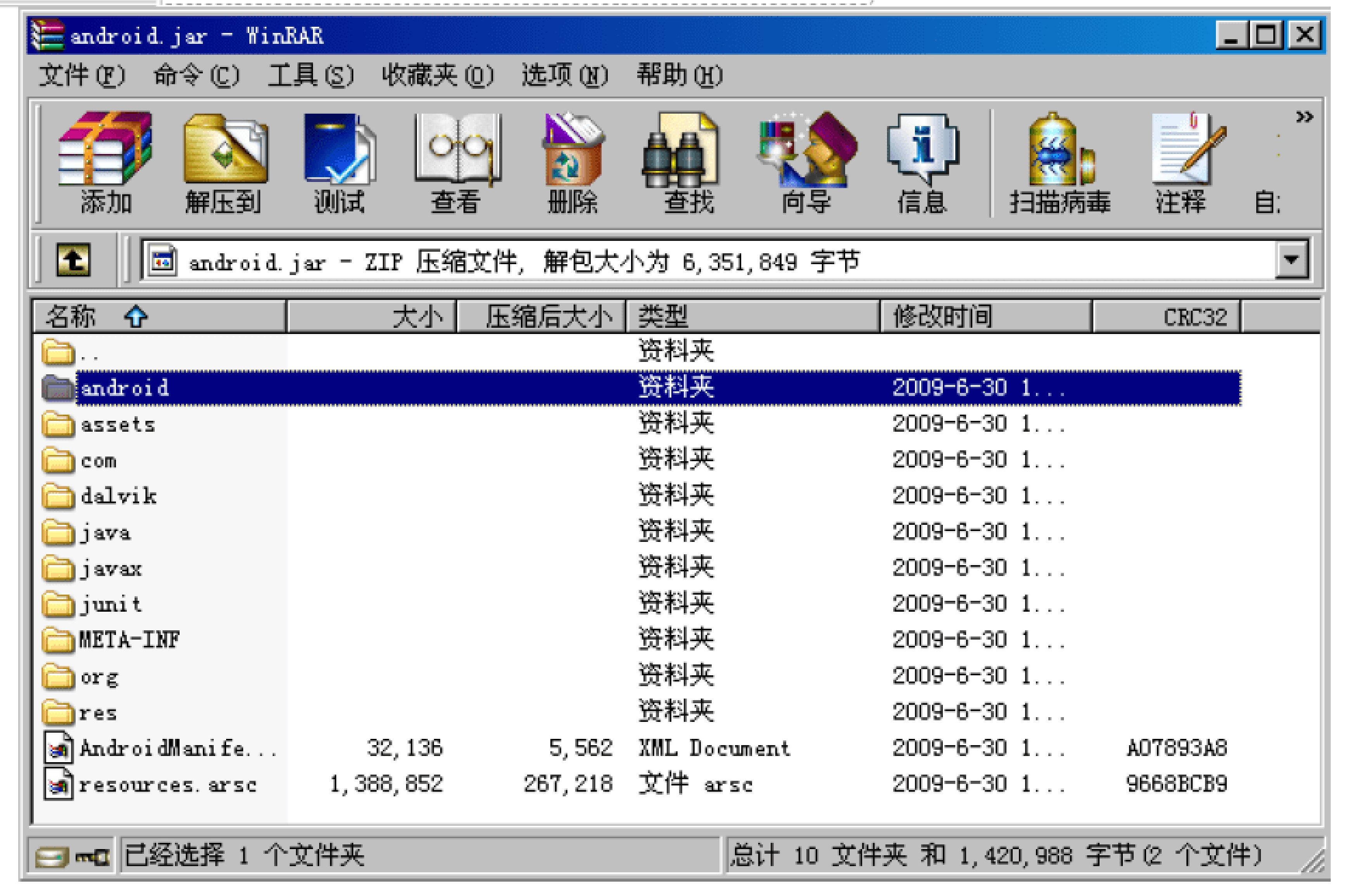


图 2-4　android.jar 文件结构(1)

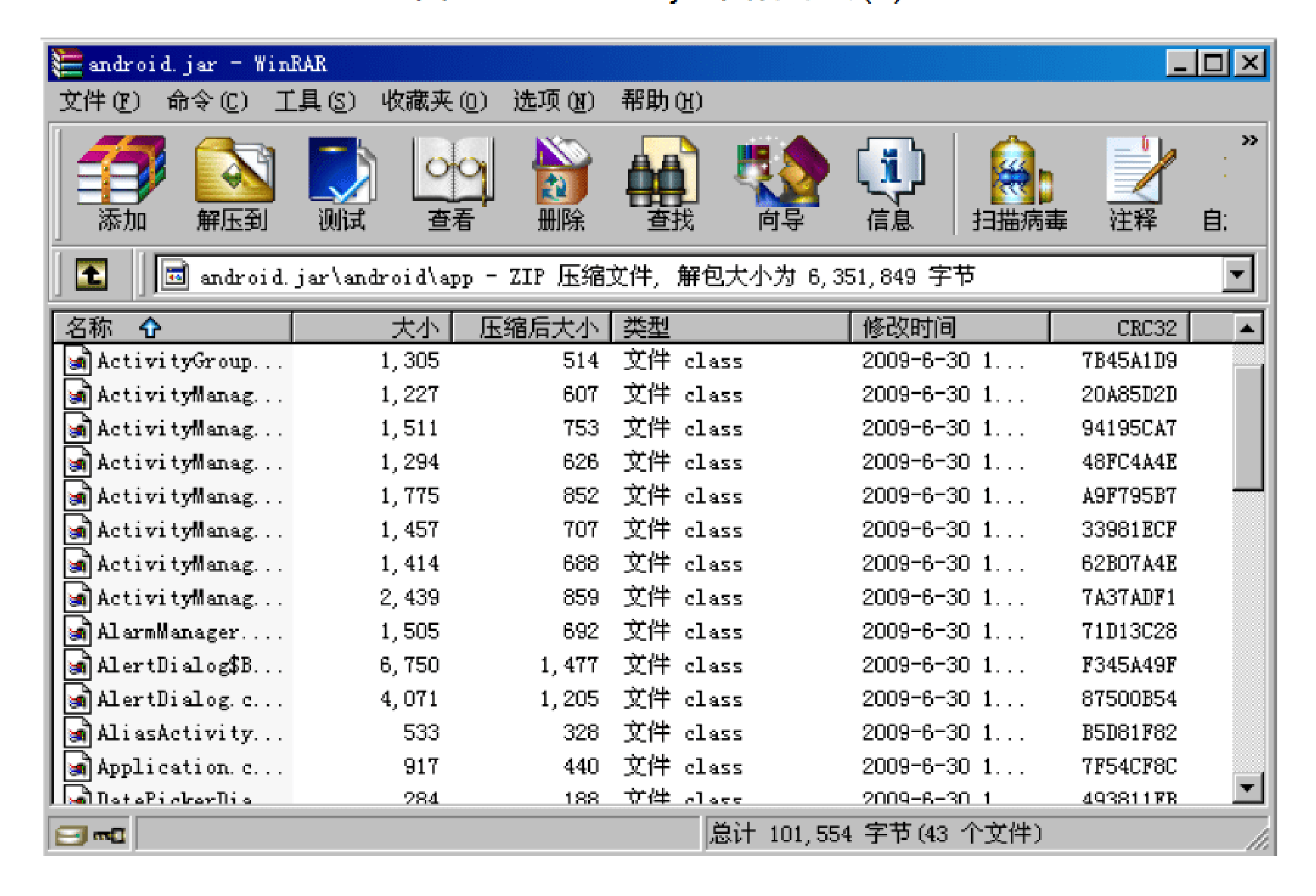


图 2-5　android.jar 文件结构(2)

2.1.3　SDK 帮助文档

要想深入理解在各个文件包内包含的 API 的具体用法，就必须学会如何阅读并查找 SDK 帮助文档信息。读者可以使用浏览器打开 docs 目录下的 index.html 文件，如图 2-6 所示。

在图 2-6 所示的主页中，介绍了 Android 的基本概念和当前的常用版本信息，并且在右侧和顶端导航栏中列出了一些常用的链接。此 SDK 文件对于初学者来说十分重要，它可以帮助读者解决很多常见的问题，是一个很好的学习文档和帮助文档。

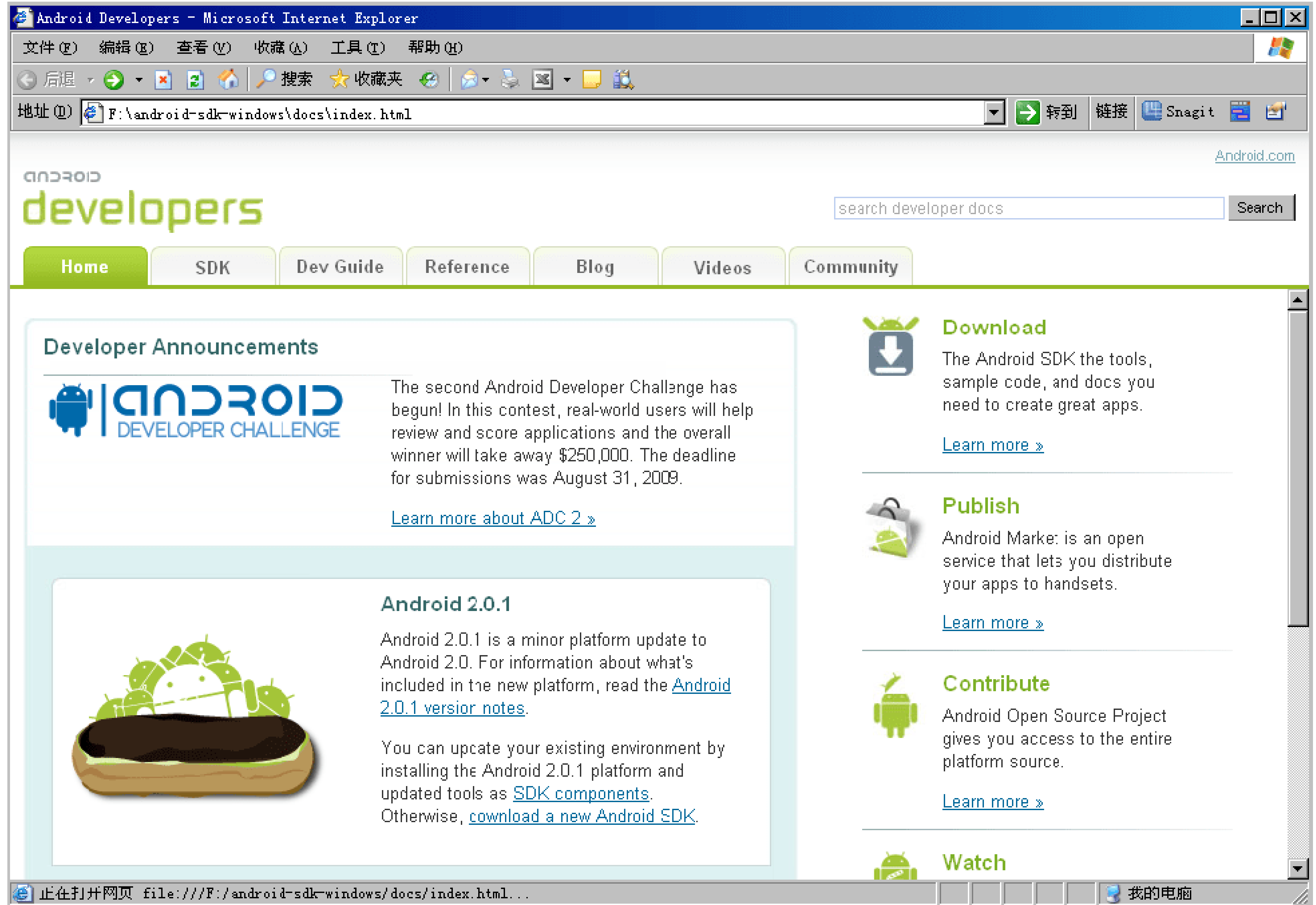


图 2-6　SDK 文档主页

单击导航中的 Dev Guide 标签，打开如图 2-7 所示的界面。

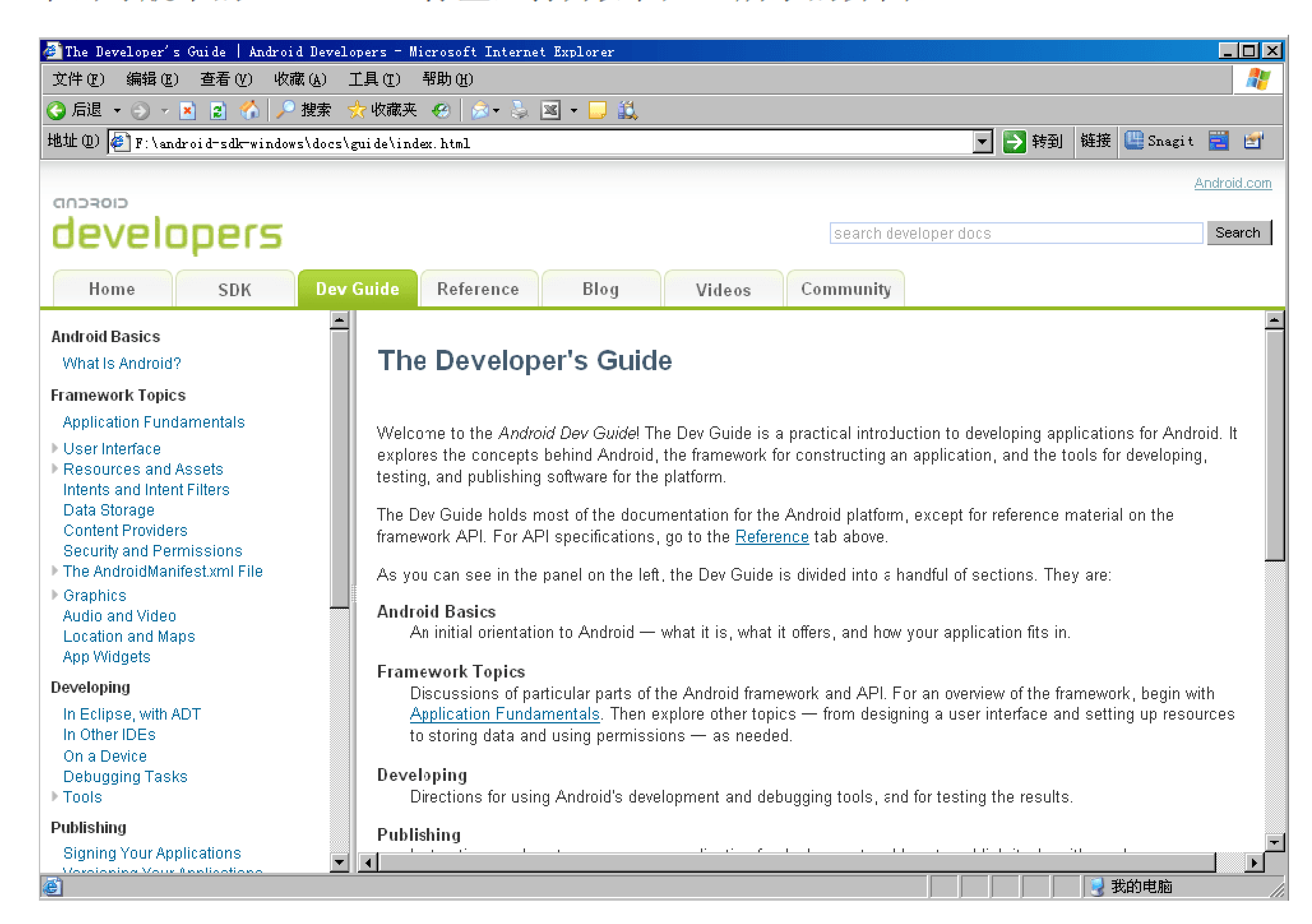


图 2-7　SDK 文档索引

图 2-7 所示页面中，左侧是目录索引链接，单击某个链接后，可以在右侧界面中显示

对应的说明信息。下面我们对各个索引目录链接进行简单的介绍。

要想迅速地理解一个问题或知识点，可以在搜索文本框中输入问题或知识点的关键字，这样可以快速地搜索到需要的知识点。当然，很多热心的程序员和学者对 SDK 进行了翻译，网络上面世了很多 SDK 中文版，感兴趣的读者可以从网络中获取。

2.1.4 Android SDK 实例简介

在 Android SDK 的安装目录中有一个名为“samples”的子目录，里面保存了几个比较具有代表性的演示实例，这些实例从不同的方面展示了 SDK 的特性和强大功能。例如里面有一个名为“JetBoy”的项目，此项目是一款具备声音支持的游戏实例，它模拟演示了如何在游戏中集成 SONiVOX 的 audioINSIDE 技术的过程，此技术是 SONiVOX 捐赠给手机联盟的。此实例可以完美地播放背景音乐和场景，实现子弹击碎飞来障碍物等一系列的效果，执行后的效果如图 2-8 所示。

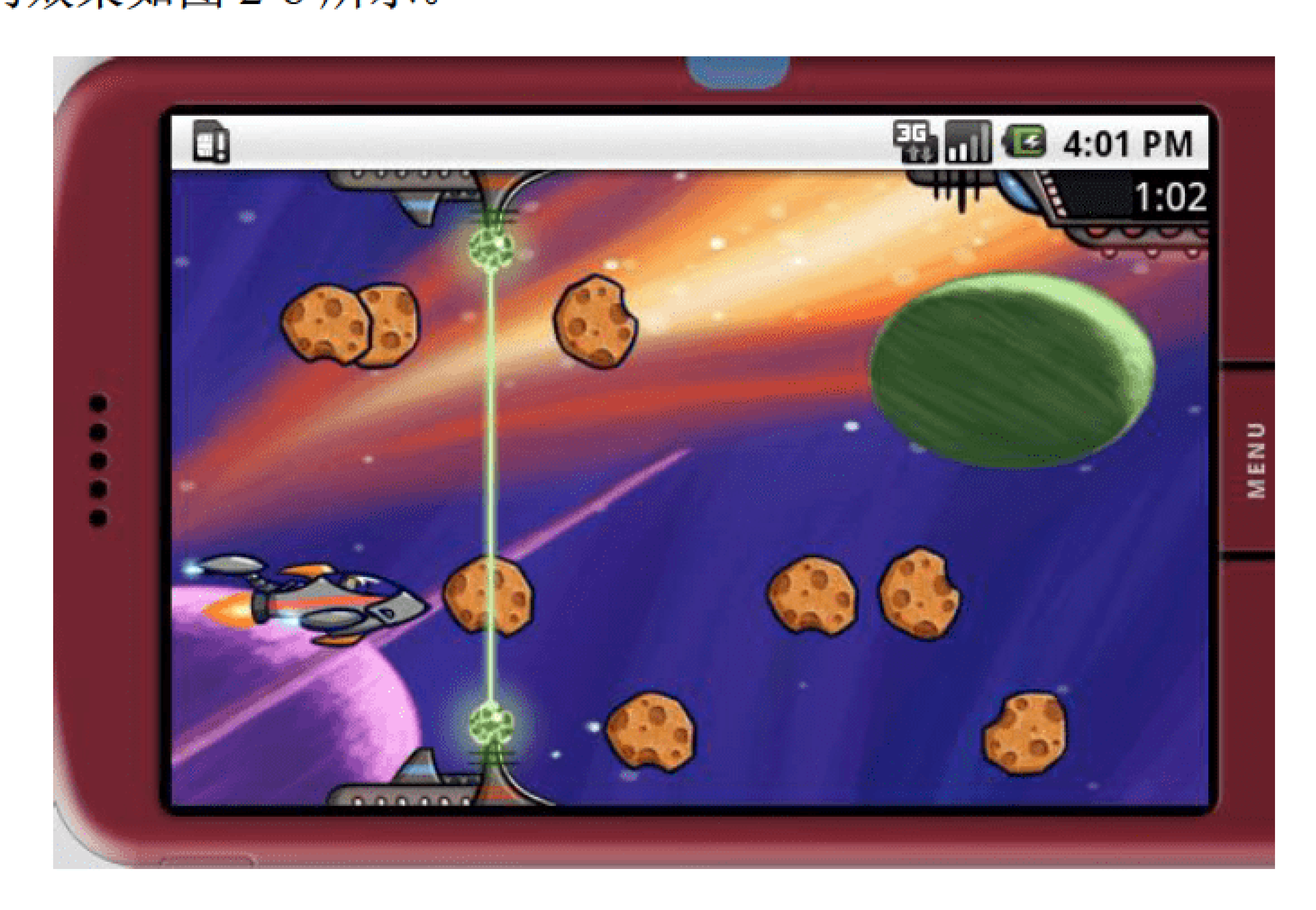


图 2-8 JetBoy 演示

2.2 Android 的系统架构详解

本节将详细讲解 Android 应用程序的核心构成部分，主要有体系结构介绍、工程文件结构和程序的生命周期等内容，为读者学习本书后面的优化知识打下基础。

2.2.1 Android 体系结构介绍

Android 作为一个移动设备的平台，其软件层次结构包括操作系统(OS)、中间件(MiddleWare)和应用程序(Application)。根据 Android 的软件框图，其软件层次结构自下而上分为以下 4 层。

(1) 操作系统层(OS)。

(2) 各种库(Libraries)和 Android 运行环境(RunTime)。



(3) 应用程序(Application)。

(4) 应用程序框架(Application Framework)。

上述各个层的具体结构如图 2-9 所示。

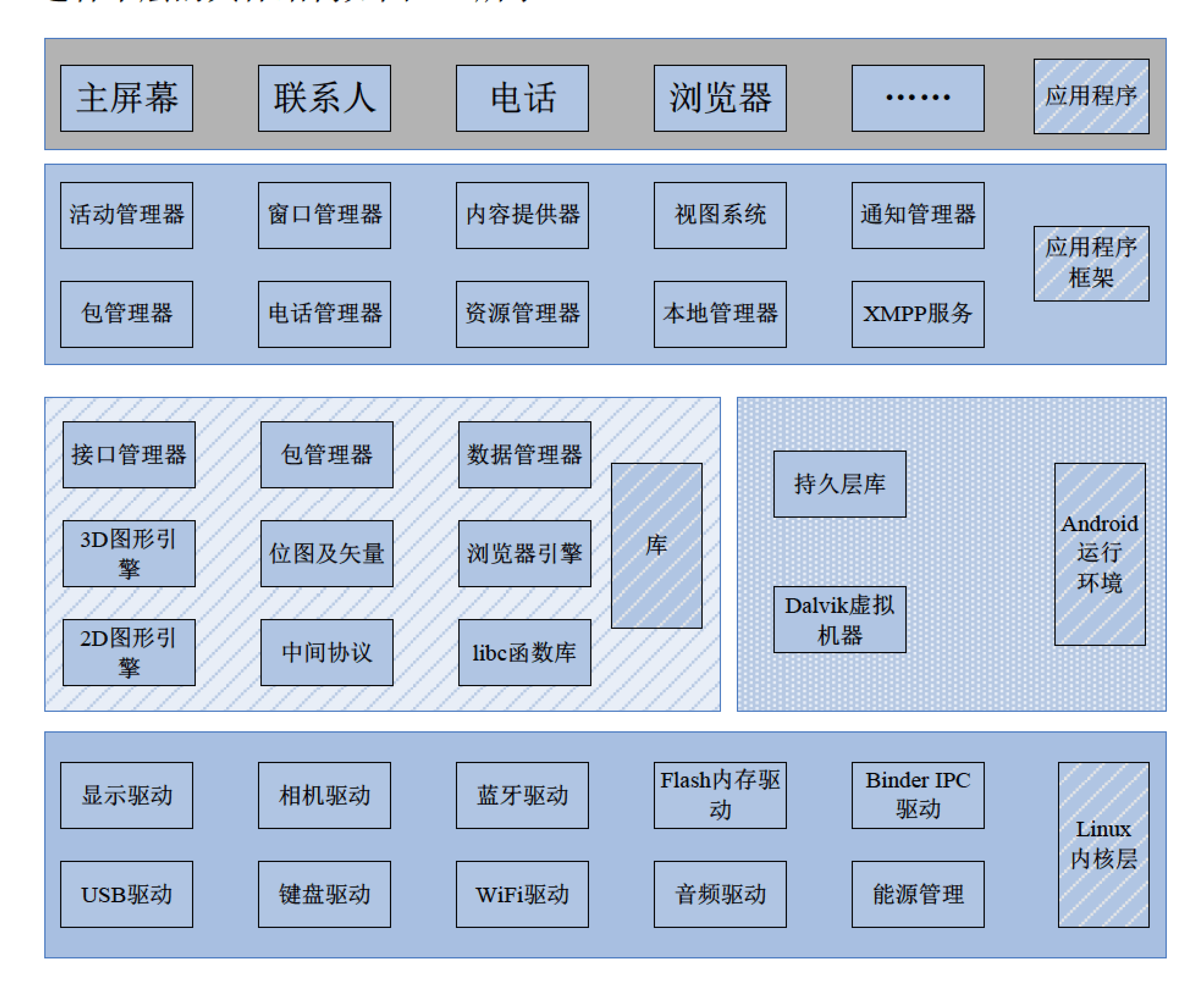


图 2-9　Android 操作系统的组件结构图

1. 操作系统层(OS)

Android 使用 Linux 2.6 作为操作系统基础，Linux 2.6 是一个开放的操作系统。Android 对操作系统的使用包括核心和驱动程序两部分。Android 的 Linux 核心为标准的 Linux 2.6 内核。Android 更多的是需要一些与移动设备相关的驱动程序，主要包含的驱动如下。

- 显示驱动(Display Driver)：常用基于 Linux 的帧缓冲(Frame Buffer)驱动。
- Flash 内存驱动(Flash Memory Driver)：是基于 MTD 的 Flash 驱动程序。
- 照相机驱动(Camera Driver)：常用基于 Linux 的 v4l(Video for Linux 的缩写)驱动。
- 音频驱动(Audio Driver)：常用基于 ALSA(Advanced Linux Sound Architecture，高级 Linux 声音体系)驱动。
- WiFi 驱动(Camera Driver)：基于 IEEE 802.11 标准的驱动程序。
- 键盘驱动(KeyBoard Driver)：作为输入设备的键盘驱动。
- 蓝牙驱动(Bluetooth Driver)：基于 IEEE 802.15.1 标准的无线传输技术。
- Binder IPC 驱动：是 Android 中一个特殊的驱动程序，具有单独的设备节点，提供进程间通信的功能。

- Power Management(能源管理)：管理电池电量等信息。

2. 各种库(Libraries)和 Android 运行环境(RunTime)

本层次对应一般嵌入式系统，相当于中间件层次。Android 的本层次分成两个部分：一部分是各种库，另一部分是 Android 运行环境。本层的内容大多是使用 C 和 C++实现的。Android 的库一般是以系统中间件的形式提供的，它们均有的一个显著特点：与移动设备平台的应用密切相关。

Android 中包含的各种库如下所示。

- C 库：是 C 语言的标准库，也是系统中的一个最为底层的库，C 库是通过 Linux 的系统调用来实现。
- 多媒体框架(MediaFrameword)：这部分内容是 Android 多媒体的核心部分，基于 PacketVideo(即 PV)的 OpenCORE，从功能上本库一共分为两大部分。一部分是音频、视频的回放(PlayBack)，另一部分则是音视频的记录(Recorder)。
- SGL：一个 2D 图像引擎。
- SSL：即 Secure Socket Layer，位于“TCP/IP”协议与各种应用层协议之间，为数据通信提供安全支持。
- OpenGL ES 1.0：提供对 3D 的支持。
- 界面管理工具(Surface Management)：提供了对管理显示子系统等功能。
- SQLite：一个通用的嵌入式数据库。
- WebKit：网络浏览器的核心。
- FreeType：表示位图和矢量字体的功能。

Android 运行环境主要是指虚拟机技术——Dalvik。Dalvik 虚拟机和一般 Java 虚拟机(Java VM)不同，它执行的不是 Java 标准的字节码(Bytecode)，而是 Dalvik 可执行格式(.dex)中的执行文件。在执行的过程中，每一个应用程序是一个进程(Linux 的一个 Process)。Dalvik 虚拟机和一般 Java 虚拟机的最大区别是：Java VM 是基于栈的虚拟机(Stack-based)，而 Dalvik 是基于寄存器的虚拟机(Register-based)。

由此可见，Dalvik 最大的好处在于可以根据硬件实现更大的优化，这更加适合移动设备的特点。

3. 应用程序(Application)

Android 方面的应用程序主要是用户界面(User Interface)方面的，通常用 Java 语言编写，其中还可以包含各种资源文件(放置在 res 目录中)。Java 程序及相关资源经过编译后，会生成一个 APK 包。Android 本身提供了主屏幕(Home)、联系人(Contact)、电话(Phone)、浏览器(Browser)等众多的核心应用。同时应用程序的开发者还可以使用应用程序框架层的 API 实现自己的程序。这也是 Android 开源的巨大潜力的体现。

4. 应用程序框架(Application Framework)

Android 的应用程序框架为应用程序层的开发者提供 APIs，它实际上是一个应用程序的框架。由于上层的应用程序是以 Java 构建的，因此本层次提供的首先包含了 UI 程序中所需要的各种控件，例如：Views(视图组件)，其中又包括了 List(列表)、Grid(栅格)、Text



Box(文本框)、Button(按钮)等，甚至一个嵌入式的 Web 浏览器。

作为一个基本的 Android 应用程序，可以利用应用程序框架中的以下 5 个部分。

(1) Activity：活动。

(2) Broadcast Intent Receiver：广播意图接收者。

(3) Service：服务。

(4) Content Provider：内容提供者。

(5) Intent and Intent Filter：意图和意图过滤器。

2.2.2　Android 工程文件结构

Android 的应用工程文件主要由以下部分组成。

- src 文件：Android 应用项目的源文件都保存在这个目录里面。
- R.java 文件：这个文件是 Eclipse 自动生成的，应用开发者不需要修改里面的内容。
- Android Library：是应用运行的 Android 库。
- assets 目录：里面主要放置多媒体等一些文件。
- res 目录：里面主要放置应用用到的资源文件。
- drawable 目录：主要放置用到的图片资源。
- layout 目录：主要放置用到的布局文件。这些布局文件都是 XML 文件。
- values 目录：主要放置字符串(strings.xml)、颜色(colors.xml)、数组(arrays.xml)。
- Androidmanifest.xml：相当于应用的配置文件。在此文件里必须声明应用的名称，应用所用到的 Activity、Service 和 receiver 等。

在 Eclipse 中，一个基本的 Android 项目的目录结构如图 2-10 所示。

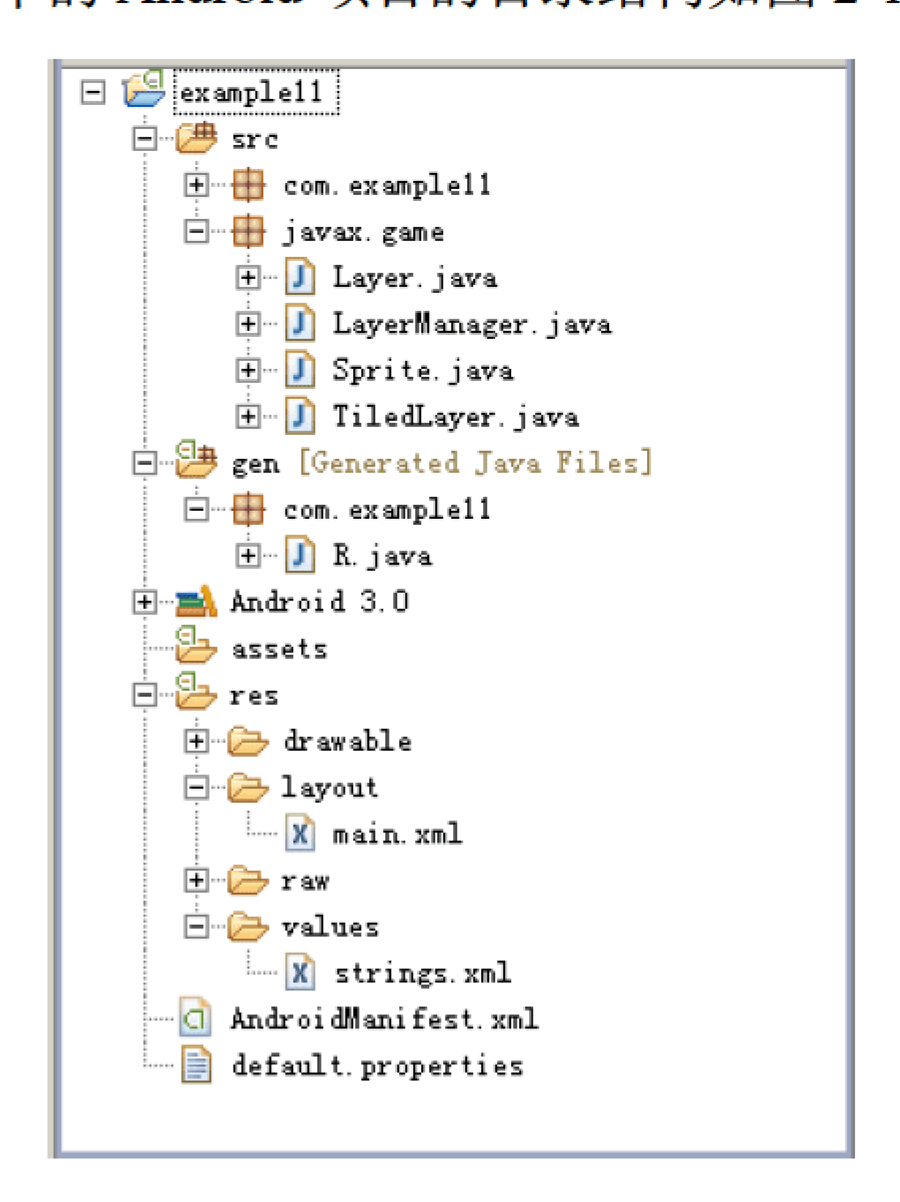


图 2-10　Android 项目的目录结构

1. src 目录

与一般的 Java 项目一样，src 目录下保存的是项目的所有包及源文件(.java)；res 目录

下包含了项目中的所有资源，例如程序图标(drawable)、布局文件(layout)和常量(values)等。不同的是，Java 项目中没有 gen 目录，也没有每个 Android 项目都必须有的文件：AndroidManfest.xml。

.java 格式文件是建立项目时自动生成的，这个文件是只读模式，不能更改。文件 R.java 是定义该项目所有资源的索引文件。例如，有一个名为“HelloAndroid”的项目，此项目中的 R.java 文件的代码如下。

```
package com.yarin.Android.HelloAndroid;
public final class R {
   public static final class attr {
   }
   public static final class drawable {
      public static final int icon=0x7f020000;
   }
   public static final class layout {
      public static final int main=0x7f030000;
   }
   public static final class string {
      public static final int app name=0x7f040001;
      public static final int hello=0x7f040000;
   }
}
```

从上述代码中可以看到定义了很多常量，并且会发现这些常量的名字都与 res 文件夹中的文件名相同，这再次证明.java 文件中所存储的是该项目所有资源的索引。有了这个文件，在程序中使用资源将变得更加方便，可以很快地找到要使用的资源，由于这个文件不能被手动编辑，所以当我们在项目中加入了新的资源时，只需要刷新一下该项目，.java 文件便会自动生成所有资源的索引。

2. 文件 AndroidManfest.xml

在文件 AndroidManfest.xml 中，包含了该项目中所使用的 Activity、Service 和 Receiver。例如，在项目“HelloAndroid”中，文件 AndroidManfest.xml 的代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
     package="com.yarin.Android.HelloAndroid"
     android:versionCode="1"
     android:versionName="1.0">
   <application android:icon="@drawable/icon"
android:label="@string/app name">
      <activity android:name=".HelloAndroid"
              android:label="@string/app_name">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
```



```
    <uses-sdk android:minSdkVersion="5" />
</manifest>
```

在上述代码中，“intent-filter”设置了启动 Activity 的位置和时间。每当一个 Activity 或操作系统要执行一个操作时将会创建出一个 Intent(具有指定作用)的对象，这个 Intent 对象承载的信息可以描述我们程序的意图，例如，想处理什么数据和什么数据类型的意图，以及一些其他信息。而 Android 会和每个 Application 所暴露的 intent-filter 的数据进行比较，直到找到最合适的 Activity 来处理调用者所指定的数据和操作为止。下面我们来仔细分析文件 AndroidManfest.xml，具体说明如表 2-1 所示。

表 2-1　AndroidManfest.xml 分析

参　数	说　明
manifest	根节点，描述了 package 中所有的内容
xmlns:android	包含命名空间的声明。xmlns:android=http://schemas.android.com/apk/res/android，使得 Android 中各种标准属性能在文件中使用，提供了大部分元素中的数据
Package	声明应用程序包
application	包含 Package 中 Application 级别组件声明的根节点。此元素也可包含 Application 的一些全局和默认的属性，如标签、icon、主题、必要的权限，等等。一个 manifest 能包含零个或一个此元素(不能大于一个)
android:icon	应用程序图标
android:label	应用程序名字
Activity	用来与用户交互的主要工具。Activity 是用户打开一个应用程序的初始页面，大部分被使用到的其他页面也由不同的 Activity 所实现，并声明在另外的 Activity 标记中。注意，每一个 Activity 必须有一个<activity>标记对应，无论它给外部使用或是只用于自己的 Package 中。如果一个 Activity 没有对应的标记，你将不能运行它。另外，为了支持运行时查找 Activity，可包含一个或多个<intent-filter>元素来描述 Activity 所支持的操作
android:name	应用程序默认启动的 Activity
intent-filter	声明了指定的一组组件支持的 Intent 值，从而形成了 Intent Filter。除了能在此元素下指定不同类型的值外，属性也能放在这里描述一个操作所需的唯一的标签、icon 和其他信息
action	组件支持的 Intent action
category	组件支持的 Intent category。这里指定了应用程序默认启动的 Activity
uses-sdk	该应用程序所使用的 SDK 版本的相关信息

3. 常量的定义文件

下面我们看看资源文件中一些常量的定义，例如文件 strings.xml 的代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="hello">Hello World, HelloAndroid!</string>
```

```
  <string name="app name">HelloAndroid</string>
</resources>
```

上述代码非常简单，只定义了两个普通的字符串资源。

接下来我们分析项目“HelloAndroid”的布局文件(layout)。首先我们打开文件res\layout\main.xml，其代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
    android:layout width="fill parent"
    android:layout height="fill parent"
    >
<TextView
    android:layout width="fill parent"
    android:layout height="wrap content"
    android:text="@string/hello"
    />
</LinearLayout>
```

在上述代码中，有以下几个布局和参数。

- <LinearLayout></LinearLayout>：线性版面配置，在这个标签中，所有元素都是按由上到下的方式排列的。
- android:orientation：表示这个介质的版面配置方式是从上到下垂直地排列其内部的视图。
- android:layout_width：定义当前视图在屏幕上所占的宽度，fill_parent 即填充整个屏幕。
- android:layout_height：定义当前视图在屏幕上所占的高度，fill_parent 即填充整个屏幕。
- wrap_content：随着文字栏位的不同而改变这个视图的宽度或高度。

在上述布局代码中，使用一个 TextView 控件配置了文本标签 Widget(构件)，其中使用属性 android:layout_width 表示整个屏幕的宽度，而属性 android:layout_height 可以根据文字来改变高度，android:text 则设置了这个 TextView 要显示的文字内容。此处引用了@string 中的 hello 字符串，即文件 string.xml 中的 hello 代表字符串资源。hello 字符串的内容可以根据自己的需要来设置，例如可以是“Hello World,HelloAndroid!”。

注意： 上面介绍的文件只是主要文件，在项目中需要我们自行编写。项目中还有很多其他的文件，那些文件需要我们编写的很少，所以在此就不进行讲解了。

2.2.3 应用程序的生命周期

程序也如同自然界中的生物一样，有自己的生命周期。应用程序的生命周期即程序的存活时间，即在什么时间内有效。Android 是一种构建在 Linux 系统之上的开源移动开发平台。在 Android 系统中，多数情况下每个程序都是在各自独立的 Linux 进程中运行的。当一个程序或其某些部分被请求时，它的进程就“出生”了。当这个程序没有必要再运行



下去，并且系统需要回收这个进程的内存用于其他程序时，这个进程就“死亡”了。由此可以看出，Android 程序的生命周期是由系统控制的，而非程序自身直接控制的。这和我们编写桌面应用程序时的思维有一些不同，一个桌面应用程序的进程也是在其他进程或用户请求时被创建，但是往往是在程序自身收到关闭请求后执行一个特定的动作(比如从 main()函数中返回)而导致进程结束的。要想做好某种类型的程序或者某种平台下的程序的开发，最关键的就是要弄清楚这种类型的程序或整个平台下的程序的一般工作模式并熟记在心。在 Android 系统中，程序的生命周期控制就属于这个范畴。

开发者必须理解不同的应用程序组件，尤其是 Activity、Service 和 Intent Receiver，特别需要了解这些组件是如何影响应用程序的生命周期的。如果不能正确地使用这些组件，可能会导致系统终止正在执行重要任务的应用程序进程。

一个常见的进程生命周期漏洞的例子是 Intent Receiver(意图接收器)，当 Intent Receiver 在方法 onReceive()中接收到一个 Intent(意图)时会启动一个线程，然后返回。一旦返回，系统将认为 Intent Receiver 不再处于活动状态，因而 Intent Receiver 所在的进程也就不再有用了，除非在该进程中还有其他的组件处于活动状态。因此，系统可能会在任意时刻终止该进程，以回收占有的内存，这样在进程中创建出的那个线程也将被终止。解决这个问题的方法是从 Intent Receiver 中启动一个服务，让系统知道在进程中还有处于活动状态的工作。为了使系统能够正确决定在内存不足时应该终止哪个进程，Android 根据每个进程中运行的组件及组件的状态把进程放入一个“Importance Hierarchy(重要性分级)”中，在其中进程的类型是按照重要程度排序的。

1. 前台进程(Foreground)

前台进程与用户当前正在做的事情密切相关。不同的应用程序组件能够通过不同的方法将它的宿主进程移到前台。在如下的任何一个条件下：进程正在屏幕的最前端运行一个与用户交互的活动(Activity)，它的 onResume 方法被调用；或进程有一个正在运行的 Intent Receiver(它的 IntentReceiver.onReceive 方法正在执行)；或进程有一个服务(Service)，并且在服务的某个回调函数(Service.onCreate、Service.onStart 或 Service.onDestroy)内有正在执行的代码，系统将把进程移动到前台。

2. 可见进程(Visible)

可见进程有一个可以被用户从屏幕上看到的活动，但不在前台(它的 onPause()方法被调用)。例如，如果前台的活动是一个对话框，以前的活动就隐藏在对话框之后，将会出现这种进程。可见进程非常重要，一般不允许被终止，除非是为了保证前台进程的运行而不得不终止它。

3. 服务进程(Service)

服务进程拥有一个已经用方法 startService()启动的服务。虽然用户无法直接看到这些进程，但它们做的事情却是用户所关心的，例如，后台 MP3 回放或后台网络数据的上传下载。因此，系统将一直运行这些进程，除非内存不足以维持所有的前台进程和可见进程。

4. 后台进程(Background)

后台进程拥有一个当前用户看不到的活动，此时它的 onStop()方法被调用。这些进程对用户体验没有直接的影响。如果它们正确地执行了活动生命周期，系统可以在任意时刻终止该进程以回收内存，并提供给前面三种类型的进程使用。系统中通常有很多这样的进程在运行，因此要将这些进程保存在 LRU 列表中，以确保当内存不足时用户最近看到的进程最后一个被终止。

5. 空进程(Empty)

空进程是不拥有任何活动的应用程序组件的进程。保留这种进程的唯一原因是在下次应用程序的某个组件需要运行时，不需要重新创建进程，这样可以提高启动速度。系统将以进程中当前处于活动状态组件的重要程度为基础对进程进行分类。进程的优先级可能也会根据该进程与其他进程的依赖关系而增长。例如，如果进程 A 通过在进程 B 中设置 Context.BIND_AUTO_CREATE 标记或使用 ContentProvider 被绑定到一个服务(Service)，那么进程 B 在分类时至少要被看成与进程 A 同等重要。

例如 Activity 的状态转换如图 2-11 所示。

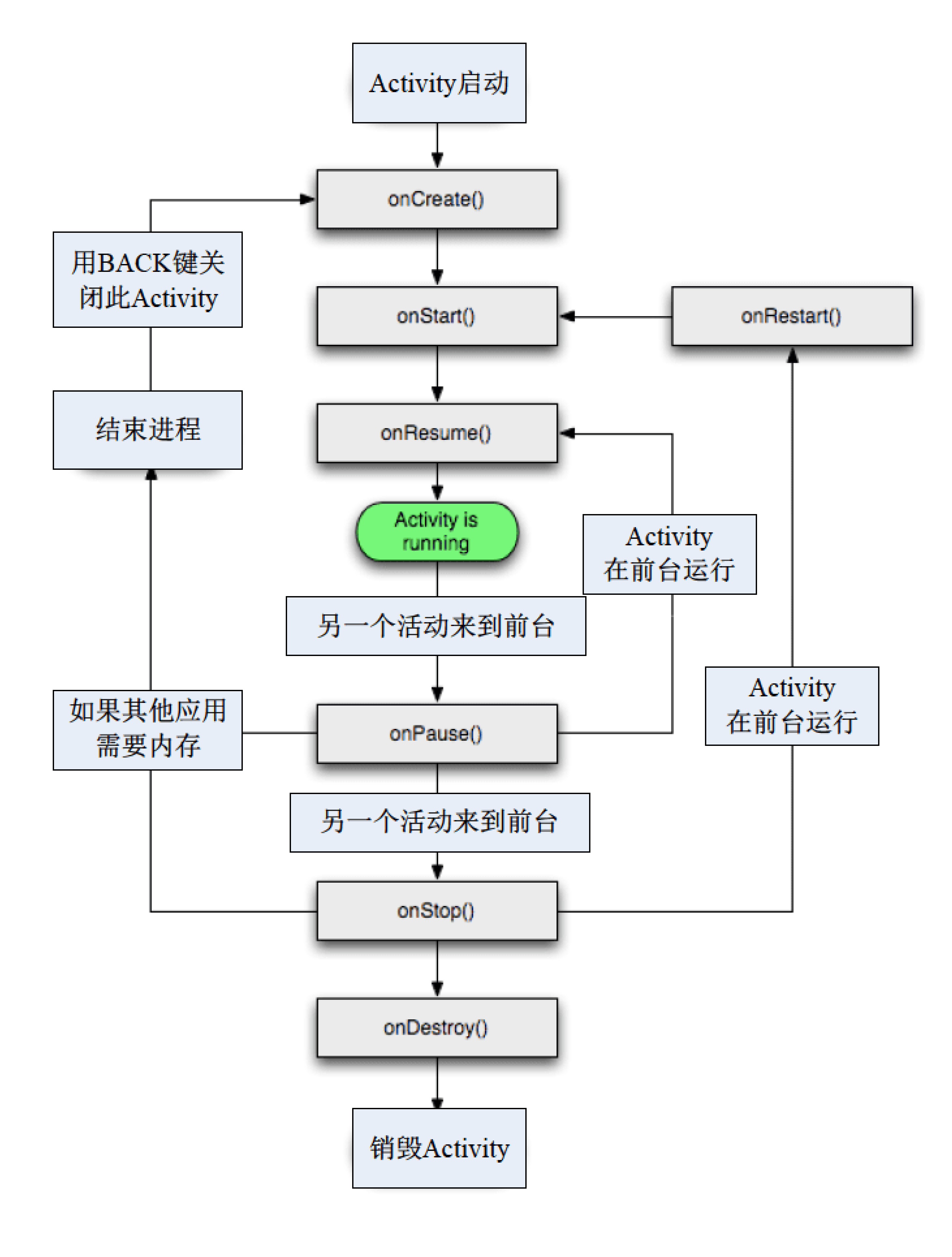


图 2-11　Activity 状态转换图



图 2-11 所示的状态变化是由 Android 内存管理器决定的，Android 会首先关闭那些包含 Inactive Activity 的应用程序，然后关闭 Stopped(已停止)状态的程序。在极端情况下，会移除 Paused(暂停)状态的程序。

2.3　简析 Android 内核

虽然本书的主要内容是讲解 Android 游戏应用开发的知识，但是为了让读者更加深入了解游戏领域的具体原理，需要介绍一些底层和内核方面的知识作为本书的铺垫。为此在本节的内容中，为读者讲解一些 Android 内核源码和驱动开发方面的知识。

2.3.1　Android 继承于 Linux

Android 是在 Linux 2.6 的内核基础之上运行的，提供的核心系统服务包括安全、内存管理、进程管理、网络组和驱动模型等内容。内核部分还相当于一个介于硬件层和系统中其他软件组之间的抽象层次。但是严格来说，它不算是 Linux 操作系统。

因为 Android 内核是由标准的 Linux 内核修改而来的，所以继承了 Linux 内核的诸多优点，保留了 Linux 内核的主体架构。同时 Android 按照移动设备的需求，在文件系统、内存管理、进程间通信机制和电源管理方面进行了修改，添加了相关的驱动程序和必要的新功能。但是和其他精简的 Linux 系统相比，例如 uClinux，Android 很大程度地保留了 Linux 的基本架构，因此 Android 的应用性和扩展性更强。

2.3.2　Android 内核和 Linux 内核的区别

Android 系统层面的底层是 Linux，并且在中间加上了一个叫作 Dalvik 的 Java 虚拟机，这从表面层看是 Android 运行库。每个 Android 应用都运行在自己的进程上，享有 Dalvik 虚拟机为它分配的专有实例。为了支持多个虚拟机在同一个设备上高效运行，Dalvik 被改写过。

Dalvik 虚拟机执行的是 Dalvik 格式的可执行文件(.dex)，该格式经过优化，以降低内存耗用到最低。Java 编译器将 Java 源文件转为.class 格式文件，.class 格式文件又被内置的 dx 工具转化为.dex 格式文件，这种文件格式可以在 Dalvik 虚拟机上注册并运行。

Android 系统的应用软件都是运行在 Dalvik 之上的 Java 软件，而 Dalvik 是运行在 Linux 中的，在一些底层功能——比如线程和低内存管理方面，Dalvik 虚拟机是依赖 Linux 内核的。由此可见，Android 是运行在 Linux 之上的操作系统，但是它本身不能算是 Linux 的某个版本。

Android 内核和 Linux 内核的差别主要体现在 11 个方面。

(1) Android Binder

Android Binder 是基于 OpenBinder 框架的一个驱动，用于提供 Android 平台的进程间通信(Inter-Process Communication，IPC)。原来的 Linux 系统上层应用的进程间通信主要是 D-bus(desktop bus)，采用消息总线的方式来进行 IPC。

其源代码位于：drivers/staging/android/binder.c。

(2) Android 电源管理(PM)

Android 电源管理是一个基于标准 Linux 电源管理系统的、轻量级的 Android 电源管理驱动，针对嵌入式设备做了很多优化。利用锁和定时器来切换系统状态，控制设备在不同状态下的功耗，以达到节能的目的。

其源代码分别位于如下文件。

- kernel/power/earlysuspend.c。
- kernel/power/consoleearlysuspend.c。
- kernel/power/fbearlysuspend.c。
- kernel/power/wakelock.c。
- kernel/power/userwakelock.c。

(3) 低内存管理器(Low Memory Killer)

Android 中的低内存管理器和 Linux 标准的 OOM(Out Of Memory)相比，其机制更加灵活，它可以根据需要杀死进程来释放需要的内存。Low Memory Killer 的代码非常简单，其核心函数是 Lowmem_shrinker()。作为一个模块在初始化时调用 register_shrinke 注册一个 lowmem_shrinker，它会被 VM 在内存紧张的情况下调用。Lowmem_shrinker 负责完成具体操作，简单来说就是寻找一个最合适的进程并杀死，从而释放这个进程所占用的内存。

其源代码位于：drivers/staging/android/lowmemorykiller.c。

(4) 匿名共享内存(Ashmem)

匿名共享内存为进程间提供大块共享内存，同时为内核提供回收和管理这个内存的机制。如果一个程序尝试访问 Kernel 释放的一个共享内存块，它将会收到一个错误提示，然后重新分配内存并重载数据。

其源代码位于：mm/ashmem.c。

(5) Android PMEM(Physical)

PMEM 用于向用户空间提供连续的物理内存区域，DSP 和某些设备只能工作在连续的物理内存上。在驱动中提供了 mmap、open、release 和 ioctl 等接口。

源代码位于：drivers/misc/pmem.c。

(6) Android Logger

Android Logger 是一个轻量级的日志设备，用于抓取 Android 系统的各种日志，是 Linux 所没有的。

其源代码位于：drivers/staging/android/logger.c。

(7) Android Alarm

Android Alarm 提供了一个定时器，用于把设备从睡眠状态唤醒，同时也提供了一个即使在设备睡眠时也会运行的时钟基准。

其源代码位于如下文件。

- drivers/rtc/alarm.c。
- drivers/rtc/alarm-dev.c。

(8) USB Gadget 驱动

此驱动是一个基于标准 Linux USB gadget 驱动框架的设备驱动，Android 的 USB 驱动

是基于 gadget 框架的。

其源代码位于如下文件。

- drivers/usb/gadget/android.c。
- drivers/usb/gadget/f_adb.c。
- drivers/usb/gadget/f_mass_storage.c。

(9) Android Ram Console

为了提供调试功能，Android 允许将调试日志信息写入一个被称为 RAM Console 的设备里，它是一个基于 RAM 的 Buffer。

其源代码位于：drivers/staging/android/ram_console.c。

(10) Android timed device

Android timed device 提供了对设备进行定时控制的功能，目前仅仅支持 vibrator 和 LED 设备。

其源代码位于：drivers/staging/android/timed_output.c(timed_gpio.c)。

(11) Yaffs2 文件系统

在 Android 系统中，采用 Yaffs2 作为 MTD NAND Flash 文件系统。Yaffs2 是一个快速稳定的应用于 NAND 和 NOR Flash 的跨平台的嵌入式设备文件系统，同其他 Flash 文件系统相比，Yaffs2 使用更小的内存来保存其运行状态，因此它占用内存小；Yaffs2 的垃圾回收非常简单而且快速，因此能达到更好的性能；Yaffs2 在大容量的 NAND Flash 上性能表现尤为明显，非常适合大容量的 Flash 存储。

其源代码位于：fs/yaffs2/目录。

2.4　简析 Android 源码

源码分析是深入掌握 Android 网络应用知识的前提工作，在本节的内容中，将简单讲解分析 Android 源码的基本知识，为读者步入本书后面知识的学习打下基础。

2.4.1　获取并编译 Android 源码

在分析 Android 源码之前，需要先下载获取 Android 源码。读者可以登录 http://source.android.com/ 获取 Android 的源码，在网页 http://source.android.com/source/downloading.html 中详细介绍了获取 Android 源码的方法，如图 2-12 所示。

在下载源码时，需要使用 repo 或 git 工具来实现。

接下来将详细介绍使用工具获取 Android 源码的流程。

(1) 创建源代码下载目录，命令如下。

```
mkdir /work/android-froyo-r2
```

(2) 用 repo 工具初始化一个版本，假如是 Android 2.2r2，则命令如下。

```
cd /work/android-froyo-r2
repo init -u git://android.git.kernel.org/platform/manifest.git -b froyo
```

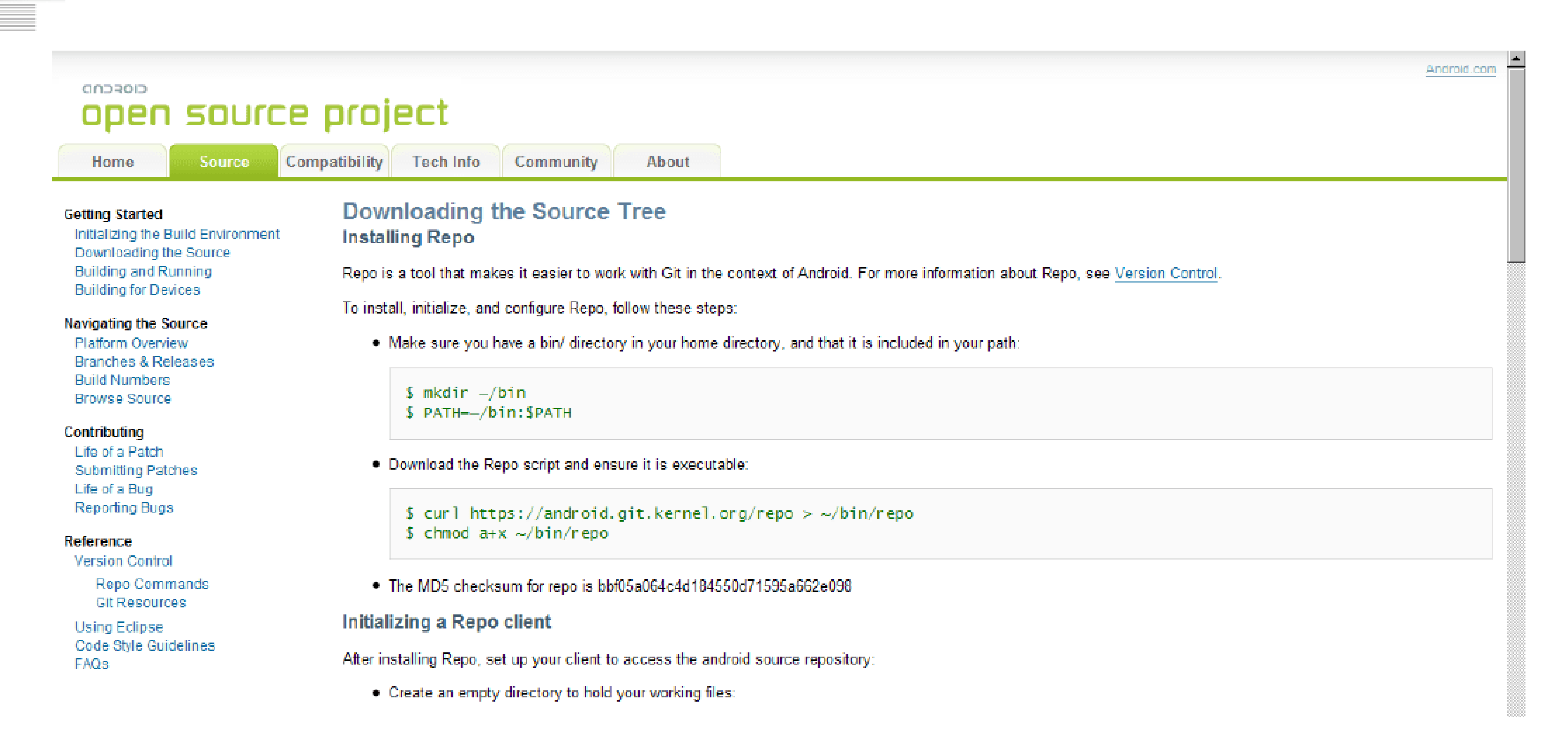


图 2-12 Linux 下获取 Android 源码的方法

在初始化过程中会显示相关版本的 TAG 信息，同时会提示我们输入用户名和邮箱地址，上面的命令初始化的是 android 2.2 froyo 的最新版本。

(3) 因为 Android 2.2 有很多个版本，这些版本信息可以从 TAG 信息中看出来。当前 froyo 的所有版本信息如下。

```
 * [new tag]         android-2.2.1 r1 -> android-2.2.1 r1
 * [new tag]         android-2.2_r1 -> android-2.2_r1
 * [new tag]         android-2.2 r1.1 -> android-2.2 r1.1
 * [new tag]         android-2.2 r1.2 -> android-2.2 r1.2
 * [new tag]         android-2.2 r1.3 -> android-2.2 r1.3
 * [new tag]         android-cts-2.2 r1 -> android-cts-2.2 r1
 * [new tag]         android-cts-2.2 r2 -> android-cts-2.2 r2
 * [new tag]         android-cts-2.2_r3 -> android-cts-2.2_r3
```

每次下载的都是最新的版本，当然也可以根据 TAG 信息下载某一特定的版本。例如下面的命令：

```
repo init -u git://android.git.kernel.org/platform/manifest.git -b
android-cts-2.3_r1
```

(4) 开始下载源码，命令如下。

```
repo sync
```

froyo 版本的代码很大，超过 2GB，下载过程非常漫长，需要读者耐心等待。

(5) 编译代码，命令如下。

```
cd /work/android-froyo-r2
make
```

上述 repo 方式获取源码的速度非常慢，其实我们完全可以通过网页浏览的方式来访问 Android 代码库，其浏览路径是 http://android.git.kernel.org/，界面如图 2-13 所示。



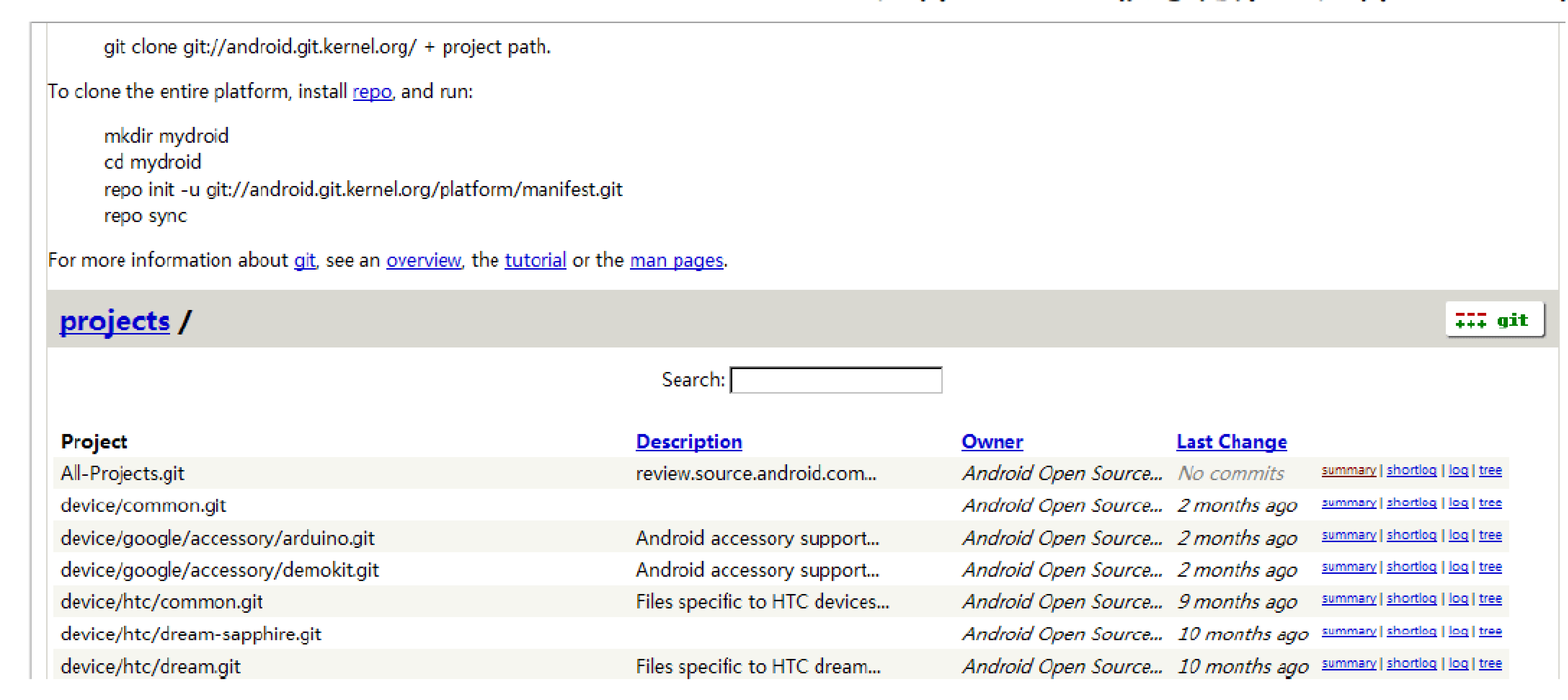


图 2-13　页面浏览方式访问 Android 代码库

注意： 因为上述获取 Android 源码的过程非常缓慢，所以一般建议不要使用 repo 来下载 Android 源码，建议直接登录 http://www.androidin.com/bbs/pub/cupcake.tar.gz 来下载，解压出来的 cupcake 中也有.repo 文件夹，此时可以通过 repo sync 来更新 cupcake 代码。获取命令如下。

```
tar -xvf  cupcake.tar.gz
```

2.4.2　Android 对 Linux 的改造

Android 内核是基于 Linux 2.6 内核的，这是一个增强的内核版本，除了修改部分 Bug 外，还提供了用于支持 Android 平台的设备驱动。Android 不但使用了 Linux 内核的基本功能，而且对 Linux 进行了改造，以实现更为强大的通信功能。

Android 中的 Linux 内核与驱动结构如图 2-14 所示。

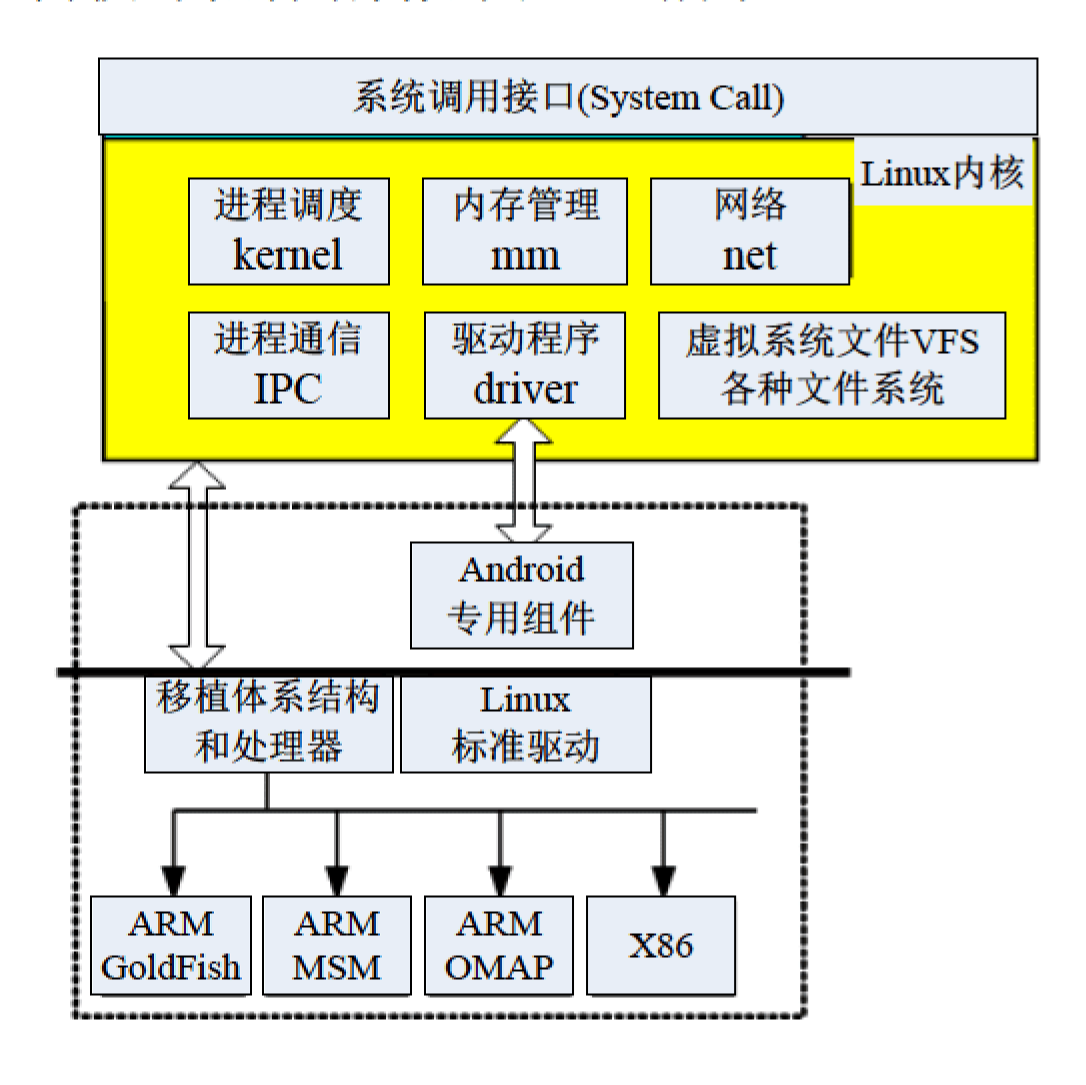


图 2-14　Android 中的 Linux 内核与驱动结构

2.4.3 为 Android 构建 Linux 的操作系统

如果我们以一个原始的 Linux 操作系统为基础，改造成为一个适合于 Android 的系统，所做的工作其实非常简单，仅仅是增加适用于 Android 的驱动程序。在 Android 系统中有很多 Linux 系统的驱动程序，将这些驱动程序移植到新系统的步骤非常简单，具体来说有以下三个步骤。

(1) 编写新的源代码。

(2) 在 KConfig 配置文件中增加新内容。

(3) 在 Makefile 中增加新内容。

在 Android 系统中，通常会使用 FrameBuffer 驱动、Event 驱动、Flash MTD 驱动、Wi-Fi 驱动、蓝牙驱动和串口等驱动程序。并且还需要音频、视频、传感器等驱动和 sysfs 接口。移植的过程就是移植上述驱动的过程，我们的工作是在 Linux 下开发适用于 Android 的驱动程序，并移植到 Android 系统中。

在 Android 中添加扩展驱动程序的基本步骤如下。

(1) 在 Linux 内核中移植硬件驱动程序，实现系统调用接口。

(2) 把硬件驱动程序的调用在 HAL 中封装成 Stub。

(3) 为上层应用的服务实现本地库，由 Dalvik 默认虚拟机调用本地库来完成上层 Java 代码的实现。

(4) 编写 Android 应用程序，提供 Android 应用服务和用户操作界面。

Android

第3章 为什么需要优化

在讲解本书的核心内容之前，通过本章让读者明白Android为什么需要优化，优化的意义是什么。希望通过本章内容的学习，能让读者充分认识到优化的迫切性，从而促使读者专心学习本节内容，为读者步入本书后面高级知识的学习打下基础。

3.1 用户体验是产品成功的关键

我们做任何一款产品，目标用户群体永远是消费者，而用户体验往往决定了一款产品的畅销程度。作为智能手机来说，因为手机的自身硬件远不及 PC，所以这就要求我们需要为消费者提供拥有更好用户体验的产品，只有这样我们的产品才会受追捧。

3.1.1 什么是用户体验

用户体验的英文称呼是 User Experience，简称为 UE。用户体验是一种纯主观在用户使用产品过程中建立起来的感受。但是对于一个界定明确的用户群体来讲，其用户体验的共性是能够经由良好设计实验来认识到。新竞争力在网络营销基础与实践中曾提到计算机技术和互联网的发展，使技术创新形态正在发生转变，以用户为中心，以人为本越来越得到重视，用户体验也因此被称作创新 2.0 模式的精髓。在中国面向知识社会的创新 2.0——应用创新园区模式探索中，更将用户体验作为“三验”创新机制之首。

1. 对用户体验的定义

权威的 ISO 9241-210 标准对用户体验的定义如下。

人们对于针对使用或期望使用的产品、系统或者服务的认知印象和回应。

由此可见，用户体验是主观的，并且其注重实际应用。另外在 ISO 定义的补充说明中，还有如下更加深入的解释。

用户体验，即用户在使用一个产品或系统之前、使用期间和使用之后的全部感受，包括情感、信仰、喜好、认知印象、生理和心理反应、行为和成就等各个方面。该说明还列出三个影响用户体验的因素，这三个因素分别是系统、用户和使用环境。

通过 ISO 标准可以推导出，可用性也可以作为用户体验的一个方面。通过可用性标准可以评估用户体验的某一些方面。不过，ISO 标准并没有进一步阐述用户体验和系统可用性之间的具体关系。由此可见，可用性和用户体验是两个相互重叠的概念。

用户体验这一领域的建立，正是为了全面地分析和透视一个人在使用某个系统时的感受。其研究重点在于系统所带来的愉悦度和价值感，而不是系统的性能。有关用户体验这一课题的确切定义、框架以及其要素还在不断发展和革新。

2. 用户体验的发展历程

“用户体验”这一名词最早在 20 世纪 90 年代中期，由用户体验设计师唐纳德•诺曼(Donald Norman)所提出和推广。在最近几年来，随着计算机技术在移动和图形技术等方面的飞速发展，已经几乎使得人机交互(HCI)技术渗透到人类活动的所有领域。这导致了一个巨大转变——(系统的评价指标)从单纯的可用性工程，扩展到范围更丰富的用户体验。这使得用户体验(用户的主观感受、动机、价值观等方面)在人机交互技术发展过程中受到了相当的重视，其关注度与传统的三大可用性指标(即效率、效益和基本主观满意度)不相上下，甚至比传统的三大可用性指标的地位更重要。



为了说明问题，我们举一个简单的例子，例如在网站设计的过程中有一点很重要，那就是需要结合不同利益相关者的利益——市场营销、品牌、视觉设计和可用性等各个方面。市场营销和品牌推广人员必须融入“互动的世界”，在这一世界里，实用性是最重要的。这就需要人们在设计网站的时候必须同时考虑到市场营销、品牌推广和审美需求这三个方面的因素。用户体验就是提供了这样一个平台，以期覆盖所有利益相关者的利益——使网站容易使用、有价值，并且能够使浏览者乐在其中。这就是为什么早期的用户体验著作都集中于网站用户体验的原因。

3.1.2　影响用户体验的因素

有许多因素可以影响用户使用系统的实际体验。为了便于讨论和分析，影响用户体验的这些因素被分为以下三大类。

- 使用者的状态。
- 系统性能。
- 环境状况。

针对典型用户群、典型环境情况的研究有助于设计和改进系统。这样的分类也有助于找到产生某种体验的原因。

3.1.3　用户体验设计目标

(1) 有用

用户体验最重要的是要让产品有用，这个有用是指用户的需求。苹果在 20 世纪 90 年代生产出第一款 PDA 手机，叫牛顿，是一个非常失败的案例。在那个年代，其实很多人并没有 PDA 的需求，苹果把 90%以上的投资放到它 1%的市场份额上，所以势必会失败。

有用这一项毋庸置疑，Android 是一款功能强大的智能手机操作系统。不但能拨打、接听电话，而且可以安装第三方软件，让手机更具有可玩性。

(2) 易用

易用是非常关键的。不容易使用的产品，也是没用的。市场上的手机有一百五十多种品牌，每一个手机有一两百种功能，当用户买到这个手机的时候，他不知道如何去用，一百多个功能他实际可能用到的就五六个。当他不理解这个产品对他有什么用，他可能就不会花钱去买这个手机。产品要让用户一看就知道怎么用，而不需要去读说明书，这也是设计的一个方向。

Android 系统集合了 Symbian、Windows 和 iOS 等系统的优点，实现每一个应用的操作都是那么的简单。并且用户可以按照自己的操作习惯进行设置，设置为符合自己操作习惯的模式。

(3) 友好

最早的时候，加入百度联盟，百度批准后，会发这样一个邮件：百度已经批准你加入百度的联盟。批准，这个语调让人非常非常难受。所以现在说：祝贺你成为百度联盟的会员。文字上的这种感觉也是用户体验的一个细节。

Android 的操作界面非常友好，UI 布局非常科学、合理，符合绝大多数人的审美习惯。

(4) 视觉设计

视觉设计的目的其实是要传递一种信息，是让产品产生一种吸引力。是这种吸引力让用户觉得这个产品可爱。“苹果”这个产品其实就有这样一个概念，就是能够让用户在视觉上受到吸引，爱上这个产品。视觉能创造出用户黏度。

Android 的视觉效果一直是用户津津乐道的，每一个颜色都凝聚了设计师的智慧结晶。

(5) 品牌

当前面 4 条做好了，就融会贯通上升到品牌。这个时候去做市场推广，可以做很好的事情。前 4 个基础没做好，推广越多，用户用得不好，他会马上走，而且永远不会再来。他还会告诉另外一个人说这个东西很难用。Android 是软件巨头谷歌公司的产品，其品牌影响力全球皆知。相信在谷歌这艘航母的承载下，Android 必然有一个美好的未来。

3.2　Android 的用户体验

Android 作为一款市场占有率排名第一的智能手机操作系统，其成功之处便是因为有良好的用户体验，不然再强大的硬件厂商支持也无济于事。在本节的内容中，将简单介绍 Android 系统自身的用户体验。

从整体看，Android 的界面美观大方，符合消费者的审美体验，如图 3-1 所示。

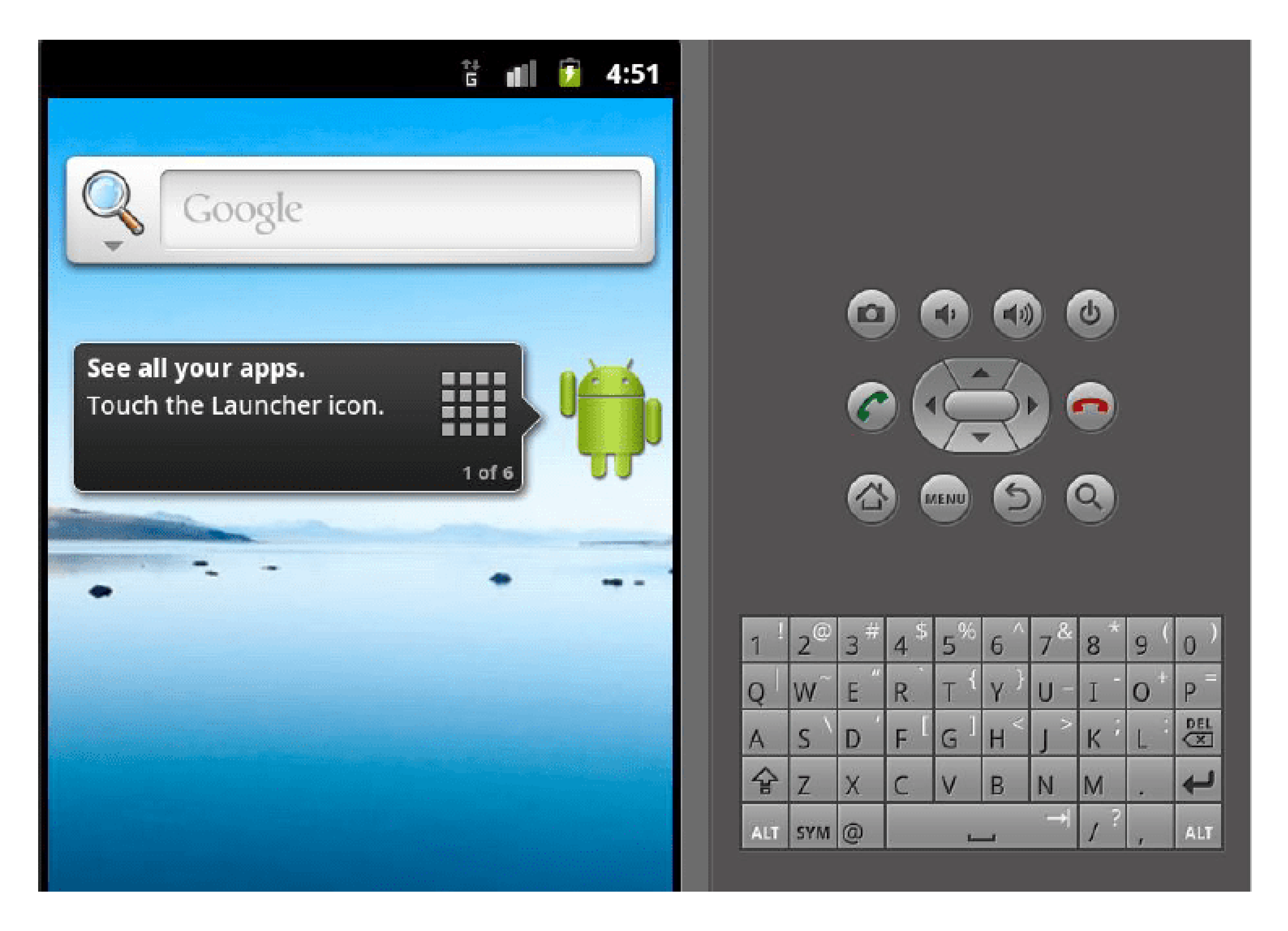


图 3-1　Android 的界面

特别是 UI(用户界面)方面，Android 用户体验团队秉承用户利益至上的原则开发。整个开发团队的精神是：当你发挥自己的创造力和思考的时候，请将它们纳入考虑之中，并有意识地加以实践。

在 Android 的官方网站中，http://developer.android.com/design/get-started/principles.html 介绍了设计 UI 的设计目标和理念，下面是笔者对这部分内容的简单理解。

(1) 以意想不到的方式取悦用户

一个漂亮的界面，一个悉心摆放的动画，或者一个适时的声音效果，都是一种快乐的体验。精细的效果能产生一种轻松的氛围，感觉手中有一股强大可控的力量，如图 3-2 所示。



图 3-2　以意想不到的方式取悦用户

(2) 真实对象比按钮和菜单更加有趣

允许人们直接触摸和操作你应用中的对象。它减少了执行一项任务所需的认识上的力量，并使之更加令人舒心，如图 3-3 所示。

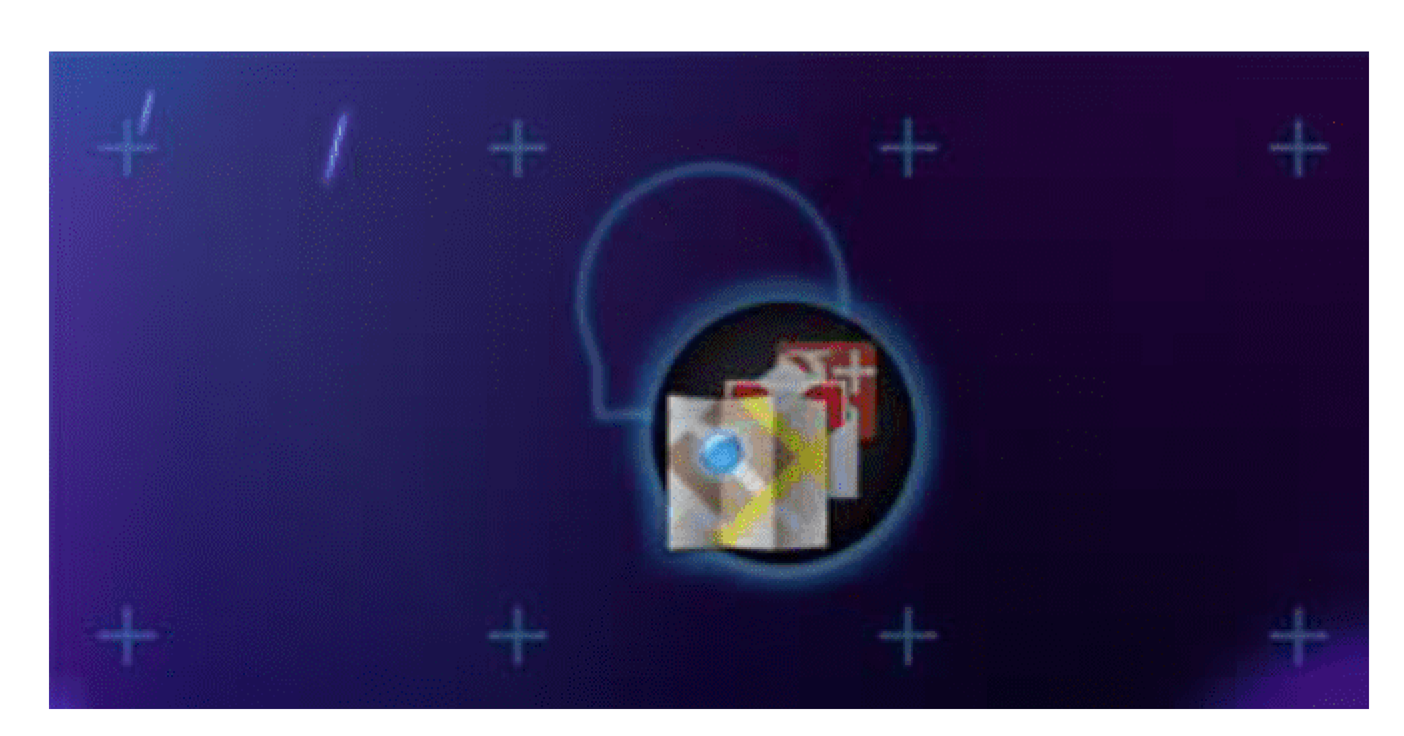


图 3-3　真实对象比按钮和菜单更加有趣

(3) 让我把它变成我的

人们喜欢加入个人手势，因为这让他们感觉自在与可控。提供可感的、漂亮的默认手势，但同时又考虑好玩、可选又不影响主要任务的定制项，如图 3-4 所示。

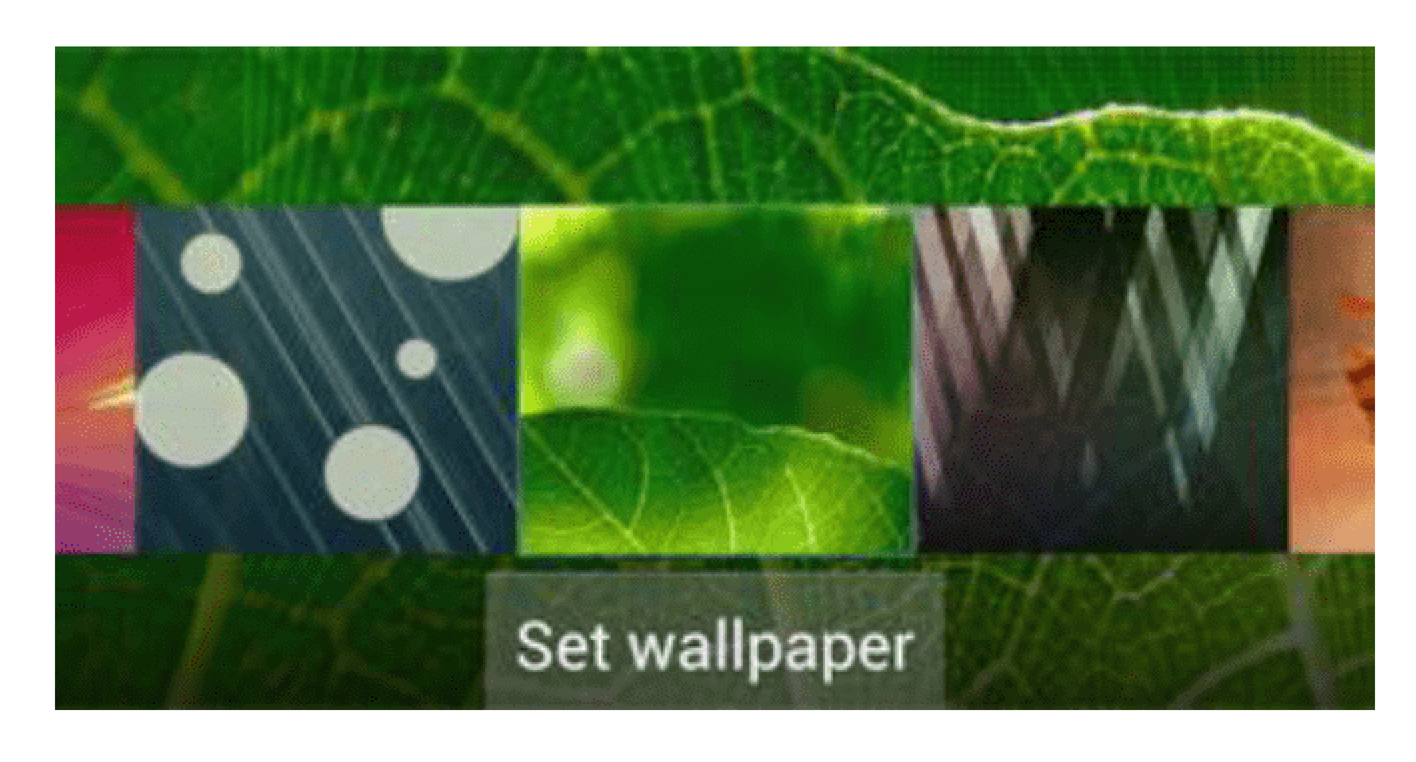


图 3-4　让我把它变成我的

(4) 学会了解我

随着时间的推移，学习用户的偏好。不要反复地问用户同样的问题，将用户先前的选择列出来以供快捷选择，如图 3-5 所示。

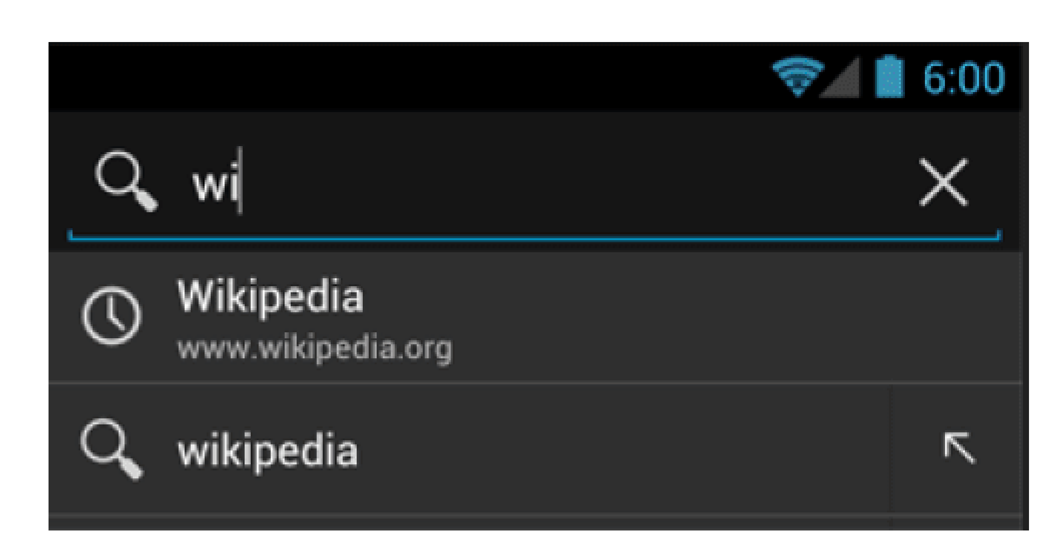


图 3-5　学会了解我

(5) 用语简洁

使用由简单词汇构成的短句。人们更倾向于跳过过长的句子，如图 3-6 所示。

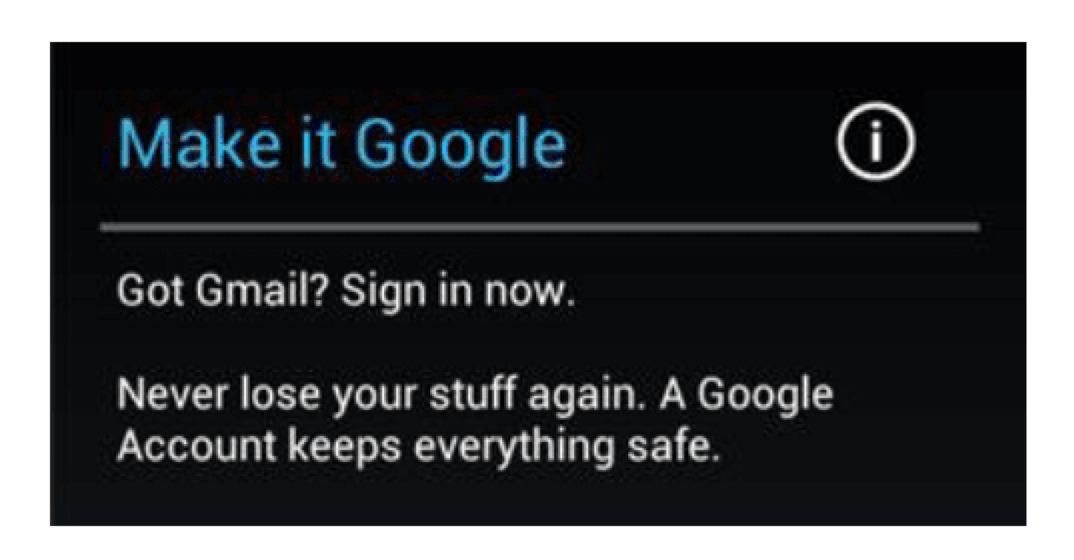


图 3-6　用语简洁

(6) 图像比文字更容易理解

考虑使用图像来解释观点。图像能捕获人们的注意力，往往比文字更有效，如图 3-7 所示。

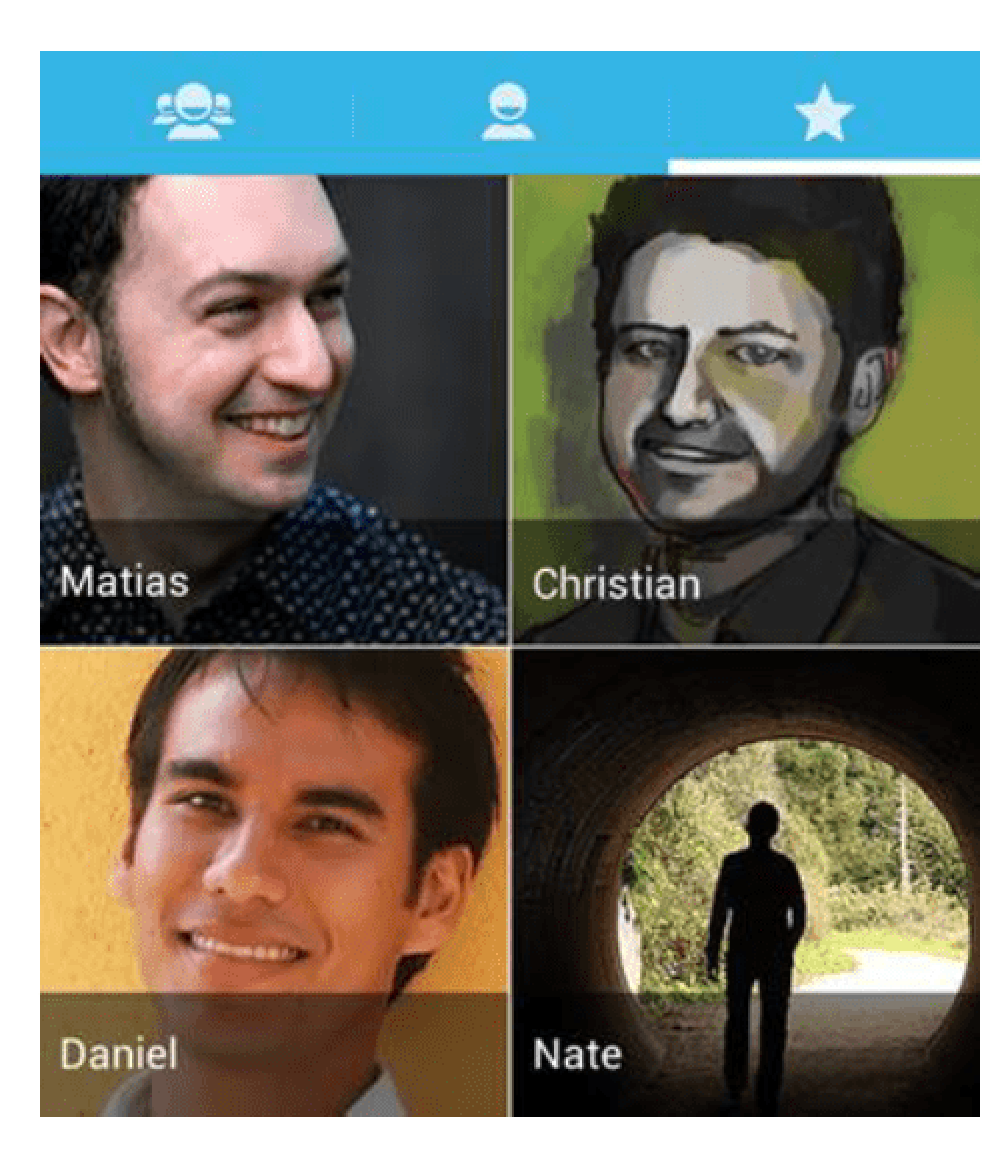


图 3-7　图像比文字更容易理解

(7) 为我决定，但最终由我说了算

做最好的猜测，先做而非先问。太多的选择和决定会令人不悦。只当你可能会犯错时，才提供个“撤销”，然后仍然先做后问，如图 3-8 所示。

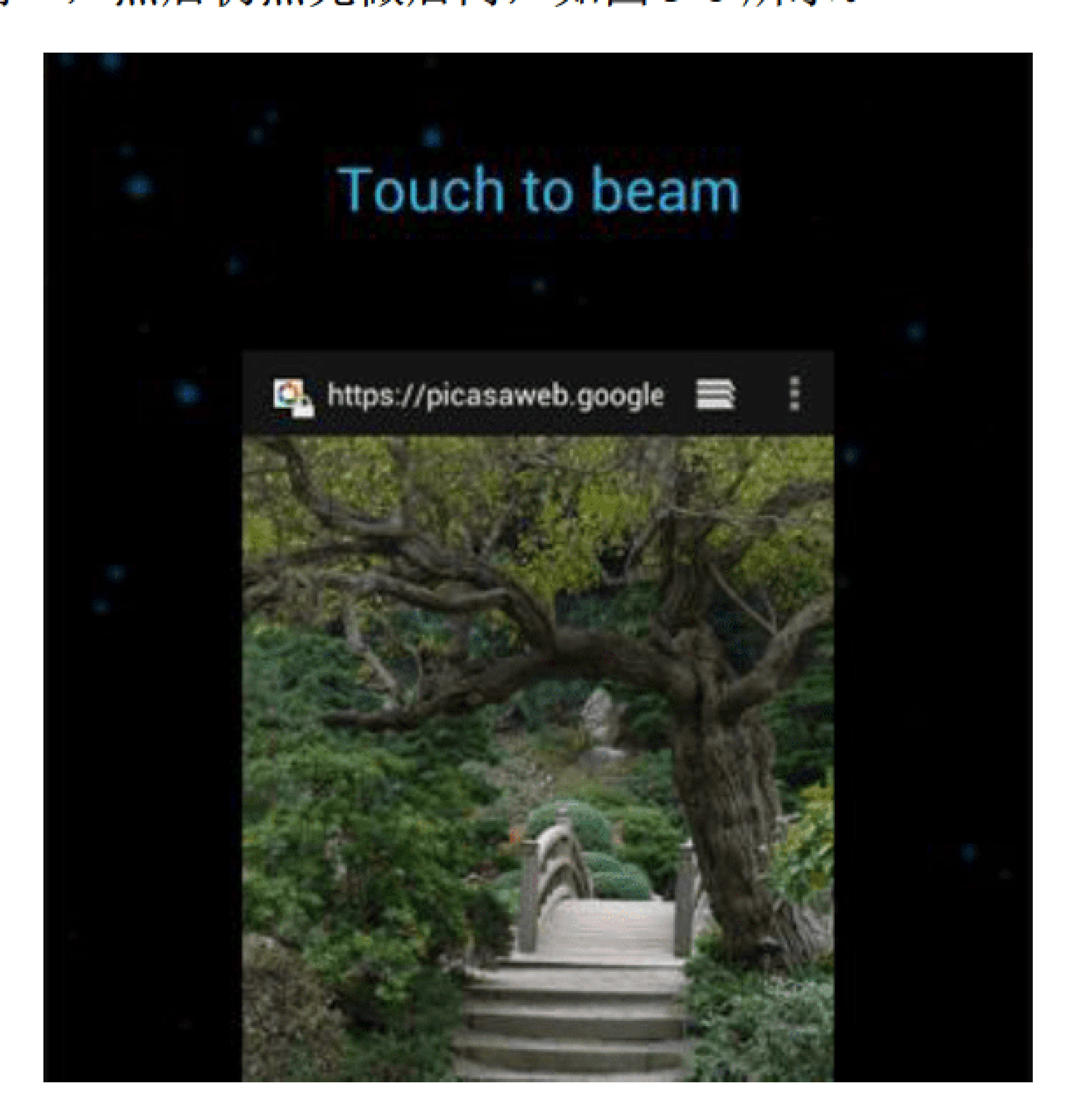


图 3-8　为我决定，但最终由我说了算

(8) 只在我需要的时候显示我所要的

当一下子看到太多东西时，人们容易受打击。将任务和信息分解成小的、可消化的片段。隐藏当前非必需的选项，并指导人们如何走下去，如图 3-9 所示。

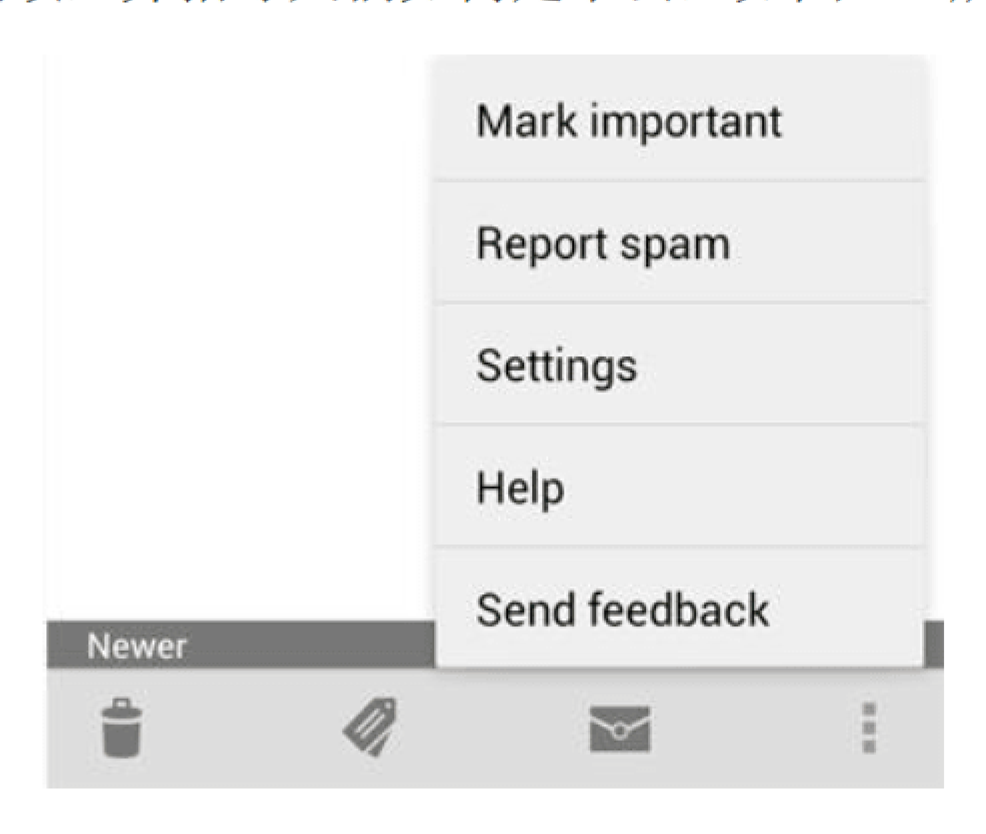


图 3-9　只在我需要的时候显示我所要的

(9) 绝不能丢失我的东西

保存用户花时间创建的东西，使得他们能随处访问。跨手机、平板电脑及计算机等平台，记住设置、个人手势以及作品，这将使得软件升级成为世界上最简单的事，如图 3-10 所示。

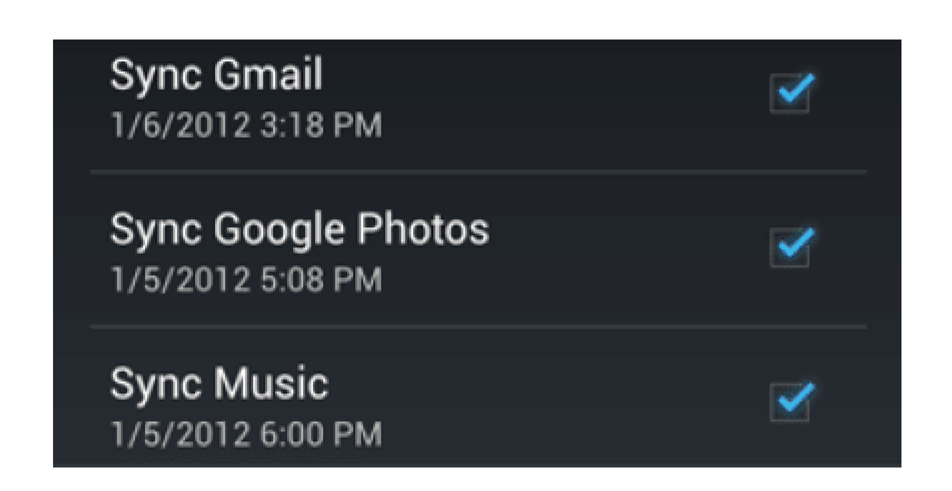


图 3-10　绝不能丢失我的东西

3.3　不同的厂商，不同的硬件

根据权威智能手机操作系统的排名，Android、iOS 是当今最受欢迎的智能手机操作系统。因为诺基亚已经基本放弃了塞班系统，而是全力和微软合作主推 Windows Phone 作为其主要的智能手机操作系统。苹果公司的 iOS 是 Android 的最大竞争对手，而苹果公司的手机产品便是 iPhone。

从有用、易用、友好、视觉设计和品牌这五个用户体验设计目标来看，iOS 和 Android 不分上下。但是从消费者的亲身体验来看，消费者的观点是 iPhone 的反应速度更加灵敏，甚至要优于硬件配置更高的 Android 机器。这是为什么呢？这得从两者的产品线说起。

1. iOS：全心全意为 iPhone 服务

iOS 是苹果公司的自有系统，而 iPhone 是苹果公司的自有手机产品，无论其 iPhone 4、iPhone 5 和 iPhone 5S，都使用 iOS 这一款系统。苹果的硬件工程师和软件工程师相互探讨，相互合作。硬件工程师的目标是选择最合理硬件来配合 iOS 系统，而软件工程师的目标是设计最合理的程序来配合硬件。并且无论是硬件工程师还是软件工程师，都对各个部分进行了优化。所以整个操作性非常灵敏，反应速度快。

2. Android：为各种机器服务

而 Android 系统恰恰相反，因为其开源和免费的特点，它不但被三星、摩托罗拉、HTC 等知名的手机厂商所用，而且也被很多山寨厂商所青睐。正是因为各种手机厂商水平的参差不齐，所以对应的硬件水平也不相同。所以 Android 的首要目标不是追求极致速度，而是追求兼容并包，要有海纳百川的胸怀供各个厂商所用。

而事实也证明了这一点，市场上三星、摩托罗拉、HTC、联想等众多品牌纷纷推出了自己的 Android 智能机，向苹果的 iPhone 发起了群狼战术。当前 Android 的目标是，用量来抢占市场，而忽视了影响用户体验的反应速度。Android 怎样才能解决这一问题呢？答案是本书的核心内容——优化。

3.4　Android 优化概述

Android 优化技术博大精深，需要程序员具备极高的水准和开发经验。笔者从事



Android 开发也是短短数载，也不可能完全掌握 Android 优化技术。本书将尽可能地将 Android 优化技术的核心内容展现给读者，希望能为读者水平的提高尽微薄之力。

本书将向大家展现如下内容。

(1) UI 布局优化

讲解了优化 UI 界面布局的基本知识，讲解了各种布局的技巧，剖析了减少层次结构、延迟加载和嵌套优化等方面的知识。

(2) 内存优化

详细讲解了 Android 系统内存的基本知识，分析了 Android 独有的垃圾回收机制，分别剖析了缩放处理、数据保存、使用与释放、内存泄漏和内存溢出等方面的知识。

(3) 代码优化

讲解了在编码过程中，优化代码提高运行效率的基本知识。

(4) 性能优化

分别讲解了资源存储、加载 DEX 文件和 APK、虚拟机的性能、平台优化、优化渲染机制等方面的知识。

(5) 系统优化

详细讲解了进程管理器、设置界面、后台停止、转移内存程序和优化缓存等方面的知识。

(6) 优化工具

详细讲解了市面中常见的优化工具，例如优化大师、进程管理等。

Android

第4章 UI布局优化

界面布局又被称为UI，UI是User Interface(用户界面)的简称。众所周知，对于网站开发人员来说，网站结构和界面设计是用户第一视觉印象的关键。而对于Android应用程序来说，除了强大的功能和方便的可操作性之外，屏幕界面效果也是影响程序质量的重要元素之一。因为消费者永远喜欢的是既界面美观，又功能强大的软件产品。在设计优美的Android界面之前，一定要先对屏幕进行布局。在布局的时候，需要用到优化技术提高界面的效率。本章将以具体实例来介绍Android系统中UI布局优化的基本知识，为读者步入本书后面知识的学习打下基础。

4.1 和布局相关的组件

在 Android 应用程序中，能用肉眼看到的内容是容易出彩的元素。这些出彩的元素都是显示在手机屏幕中的，但是手机屏幕十分有限，究竟怎样摆放才能更加优美是 UI 布局所负责的任务。在看似简单的手机界面中，不是随随便便简单布置实现元素排列的，而是使用 UI 组件实现界面布局的。

4.1.1 View 视图组件

在 Android 系统中，类 View 是一个最基本的 UI 类，几乎所有的 UI 组件都是继承于 View 类而实现的。类 View 的主要功能如下。

(1) 为指定的屏幕矩形区域存储布局和内容。

(2) 处理尺寸和布局、绘制、焦点改变、翻屏、按键、手势。

(3) widget 基类。

类 View 的语法格式如下。

```
Android.view.View
```

在 Android 中常用的 View 子类如表 4-1 所示。

表 4-1 View 类

文本(TextView)	输入框(EditText)
输入法(InputMethod)	活动方法(MovementMethod)
复选框(Checkbox)	滚动视图(ScrollView)
按钮(Button)	单选按钮(RadioButton)

4.1.2 Viewgroup 容器

Viewgroup 仿佛是一个容器，我们可以对它里面的 View 进行布局处理。使用 Viewgroup 的语法格式如下。

```
android.view.Viewgroup
```

Viewgroup 能够包含并管理下级系列的 Views 和其他 Viewgroup，它是一个布局的基类。Viewgroup 好像一个 View 容器，负责对添加进来的 View 进行布局处理。在一个 Viewgroup 中可以看见另一个 Viewgroup 中的内容。各个 Viewgroup 类之间的关系如图 4-1 所示。

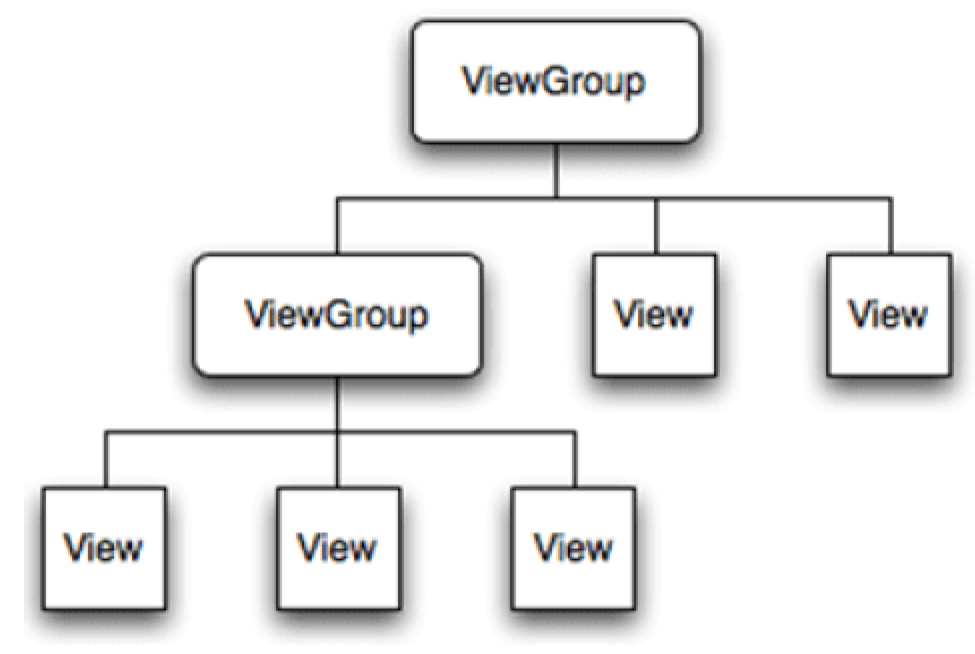


图 4-1　各类的继承关系

4.2　Android 中的 5 种布局方式

在 Viewgroup 里面可以装下很多控件，我们布局的作用就是对这些控件进行排列，排列成最实用的效果。在布局里面还可以套用其他的布局，这样可以实现界面多样化以及设计的灵活性。使用布局组件 Layout 的语法格式如下。

```
<LinearLayout xmlns:Android="http://schemas.Android.com/apk/res/Android"
   Android:orientation="vertical"
   Android:layout width="fill parent"
   Android:layout height="fill parent"
>
```

在一个布局容器里可以包括零个或多个布局容器，在 Android 中有 5 种界面布局对象，分别是 FrameLayout(框架布局)、LinearLayout(线性布局)、AbsoluteLayout(绝对布局)、RelativeLayout(相对布局)、TableLayout(表格布局)。本节将详细讲解这 5 种布局对象的基本知识和使用方法。

4.2.1　线性布局 LinearLayout

线性布局 LinearLayout 能够根据为它设置的垂直或水平属性值来排列所有的子元素。所有的子元素都被堆放在其他元素之后，因此一个垂直列表的每一行只会有一个元素，而不管它们有多宽，而一个水平列表将会只有一个行高(高度为最高子元素的高度加上边框高度)。LinearLayout 保持子元素之间的间隔以及互相对齐(相对一个元素的右对齐、中间对齐或者左对齐)。

LinearLayout 还支持为单独的子元素指定 weight，这样的好处是允许子元素可以填充屏幕上的剩余空间。同时也避免了在一个大屏幕中，一串小对象挤成一堆的情况，可以允许它们放大填充空白。子元素指定一个 weight 值，剩余的空间就会按这些子元素指定的 weight 比例分配给这些子元素。默认的 weight 值为 0。假设有三个文本框，其中两个指定了 weight 值为 1，那么，这两个文本框将等比例地放大，并填满剩余的空间，而第三个文本框不会放大。

通过 LinearLayout 线性布局，可以在一个方向上(垂直或水平)对齐所有子元素。在里面既可以将所有子元素罗列堆放，也可以一个垂直列表每行将只有一个子元素(无论它们有多宽)，如图 4-2 所示。另外也可以一个水平列表只是一列的高度，如图 4-3 所示。



图 4-2　垂直布局

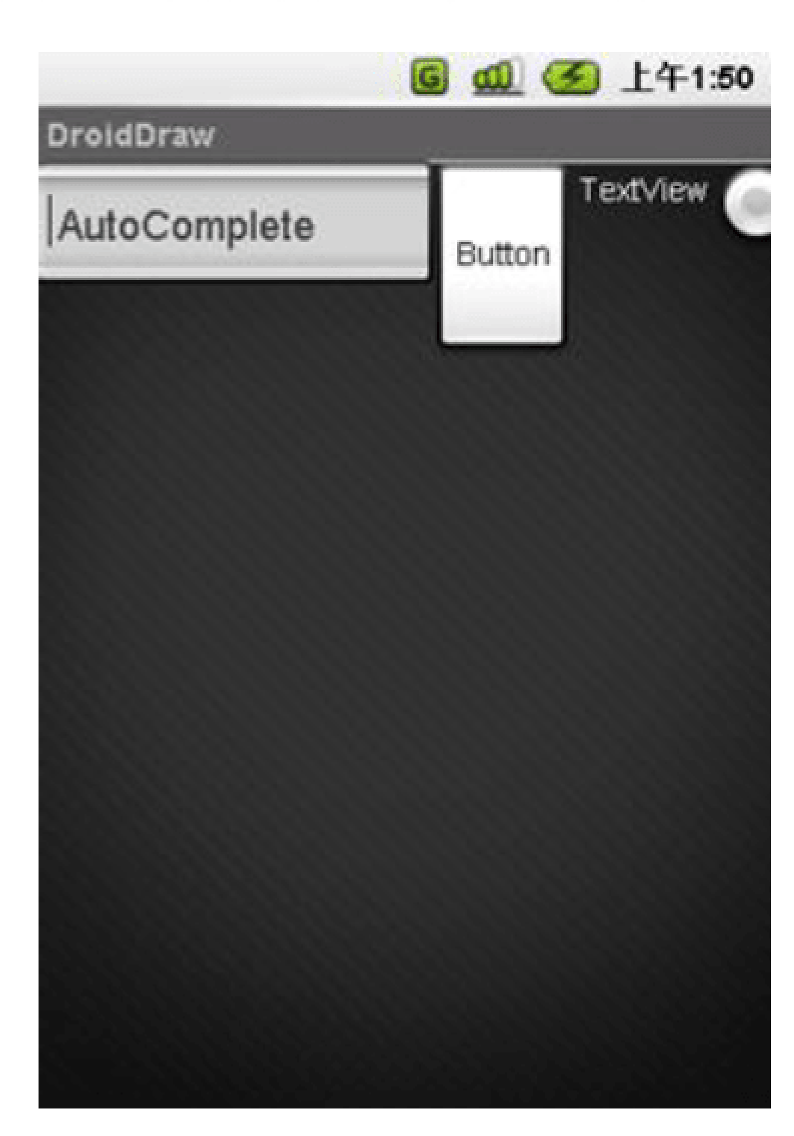


图 4-3　水平布局

下面用事实来说话，通过实例来说明线性布局的用法。

实例 1	
源码路径	\daima\4\vertical
功能	演示垂直线性布局的用法

本实例的具体实现流程如下。

(1) 打开 Eclipse，依次选择 File | New | Android Project 命令，新建一个名为“vertical”的工程文件。

(2) 编写布局文件 main.xml，在里面插入 4 个 TextView，分别用于显示“这是第 1 行”、“这是第 2 行”、“这是第 3 行”和“这是第 4 行”共 4 行文本。具体代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
    <TextView
       android:text="这是第 1 行"
       android:gravity="center_vertical"
       android:textSize="15pt"
       android:background="#aa0000"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:layout_weight="1"/>
```

 UI 布局优化

```
    <TextView
        android:text="这是第 2 行"
        android:textSize="15pt"
        android:gravity="center vertical"
        android:background="#00aa00"
        android:layout width="fill parent"
        android:layout height="wrap content"
        android:layout weight="1"/>
    <TextView
        android:text="这是第 3 行"
        android:textSize="15pt"
        android:gravity="center vertical"
        android:background="#0000aa"
        android:layout width="fill parent"
        android:layout height="wrap content"
        android:layout weight="1"/>
    <TextView
        android:text="这是第 4 行"
        android:textSize="15pt"
        android:gravity="center vertical"
        android:background="#aaaa00"
        android:layout width="fill parent"
        android:layout height="wrap content"
        android:layout weight="1"/>
</LinearLayout>
```

(3) 编写文件 strings.xml 设置项目中的文本值，主要代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">垂直</string>
    <string name="app name">垂直</string>
</resources>
```

(4) 主文件 Activity01.java 是自动生成的，能够调用界面布局文件的样式在手机屏幕中显示，主要代码如下。

```
public class Activity01 extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

到此为止，整个编码工作全部结束，执行效果如图 4-4 所示。



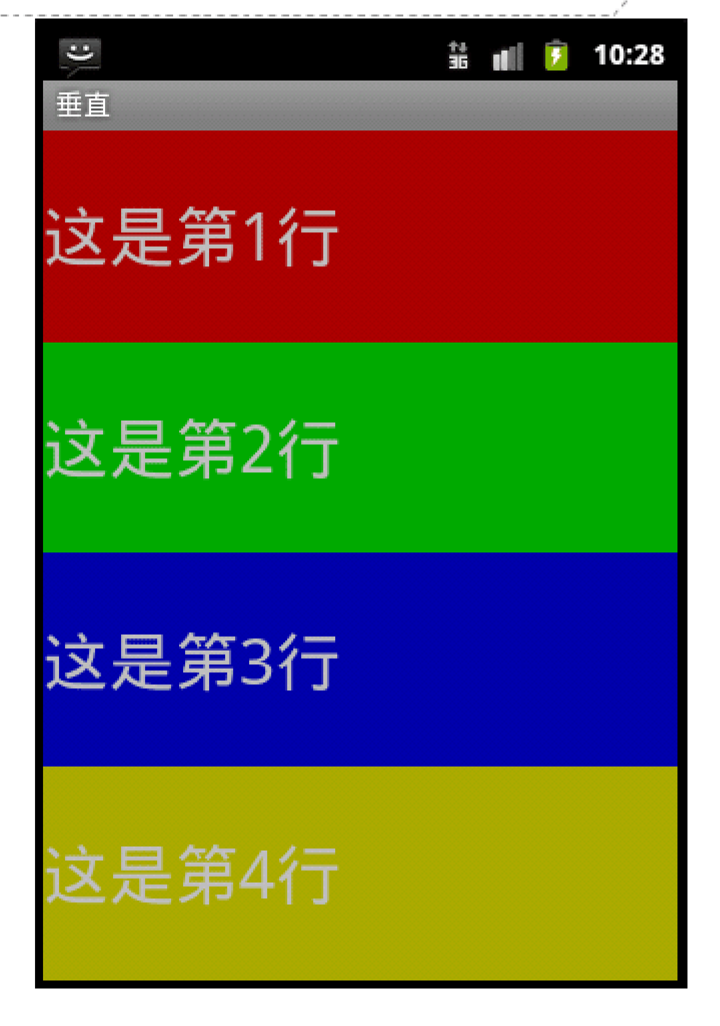


图 4-4　执行效果

4.2.2　框架布局 FrameLayout

框架布局 FrameLayout 是 Android 中最简单的一个布局对象，它被定制为屏幕上的一个空白备用区域，之后可以在里面填充一个单一对象，例如一张图片。FrameLayout 默认将所有的子元素固定在屏幕的左上角，我们不能设置 FrameLayout 中一个子元素的位置。后一个子元素将会直接在前一个子元素之上进行覆盖填充，把它们部分或全部挡住(除非后一个子元素是透明的)。看下面的一段代码：

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout width="fill parent"
   android:layout height="fill parent"
   >
 <!-- 我们在这里加了一个 Button 按钮 -->
<Button
   android:text="button"
   android:layout width="fill parent"
   android:layout height="wrap content"
/>
<TextView
   android:text="textview"
   android:textColor="#0000ff"
   android:layout width="wrap content"
   android:layout height="wrap content"
/>
</FrameLayout>
```

上述代码使用了 FrameLayout 布局，执行后的效果如图 4-5 所示。

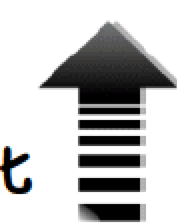




图 4-5 FrameLayout 布局

4.2.3 绝对布局 AbsoluteLayout

绝对布局 AbsoluteLayout 可以指定其子元素的准确的 x、y 坐标位置，并显示在屏幕上。用(0，0)表示左上角，当向下或向右移动时坐标值将随之变大。AbsoluteLayout 没有页边框，可以允许元素之间互相重叠。看下面的一段代码：

```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout width="fill parent"
    android:layout height="fill parent"
    >
<EditText
    android:text="Welcome to Mr Wei's blog"
    android:layout width="fill parent"
    android:layout height="wrap content"
/>
<Button
    android:layout x="250px"                //设置按钮的 X 坐标
    android:layout y="40px"                 //设置按钮的 Y 坐标
    android:layout width="70px"             //设置按钮的宽度
    android:layout height="wrap content"
    android:text="Button"
/>
</AbsoluteLayout>
```

上述代码使用了 AbsoluteLayout 布局，执行后的效果如图 4-6 所示。



图 4-6 AbsoluteLayout 布局

注意： 在日常项目应用中不推荐使用 AbsoluteLayout，因为它会使界面代码太过刚性，以至于在不同的设备上会存在不能很好地工作的情况。

4.2.4 相对布局 RelativeLayout

相对布局 RelativeLayout 指定子元素相对于其他元素或父元素的位置(通过 ID 指定)。

我们可以以右对齐、上下或置于屏幕中央的形式来排列两个元素。因为元素按顺序排列，因此如果第一个元素在屏幕的中央，那么相对于这个元素的其他元素将以屏幕中央的相对位置来排列。如果使用 XML 来指定这个 layout，在定义它之前必须定义被关联的元素。看下面的一段代码：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView
      android:id="@+id/label"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Welcome to Mr Wei's blog:"/>
   <EditText
      android:id="@+id/entry"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_below="@id/label"/>
   <Button
      android:id="@+id/ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_below="@id/entry"
      android:layout_alignParentRight="true"
      android:layout_marginLeft="10dip"
      android:text="OK" />
   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_toLeftOf="@id/ok"
      android:layout_alignTop="@id/ok"
      android:text="Cancel" />
</RelativeLayout>
```

上述代码使用了 RelativeLayout 布局，执行后的效果如图 4-7 所示。



图 4-7　RelativeLayout 布局

在 RelativeLayout 布局方式中，允许子元素指定它们相对于其他元素或父元素的位置(通过 ID 指定)。我们可以以右对齐、上下或置于屏幕中央的形式来排列两个元素。在 RelativeLayout 中的元素是按顺序排列的，如果第一个元素在屏幕的中央，那么相对于这个元素的其他元素将以屏幕中央的相对位置来排列。如果使用 XML 来指定这个 layout，那么在定义它之前必须定义被关联的元素。其结构说明如图 4-8 所示。

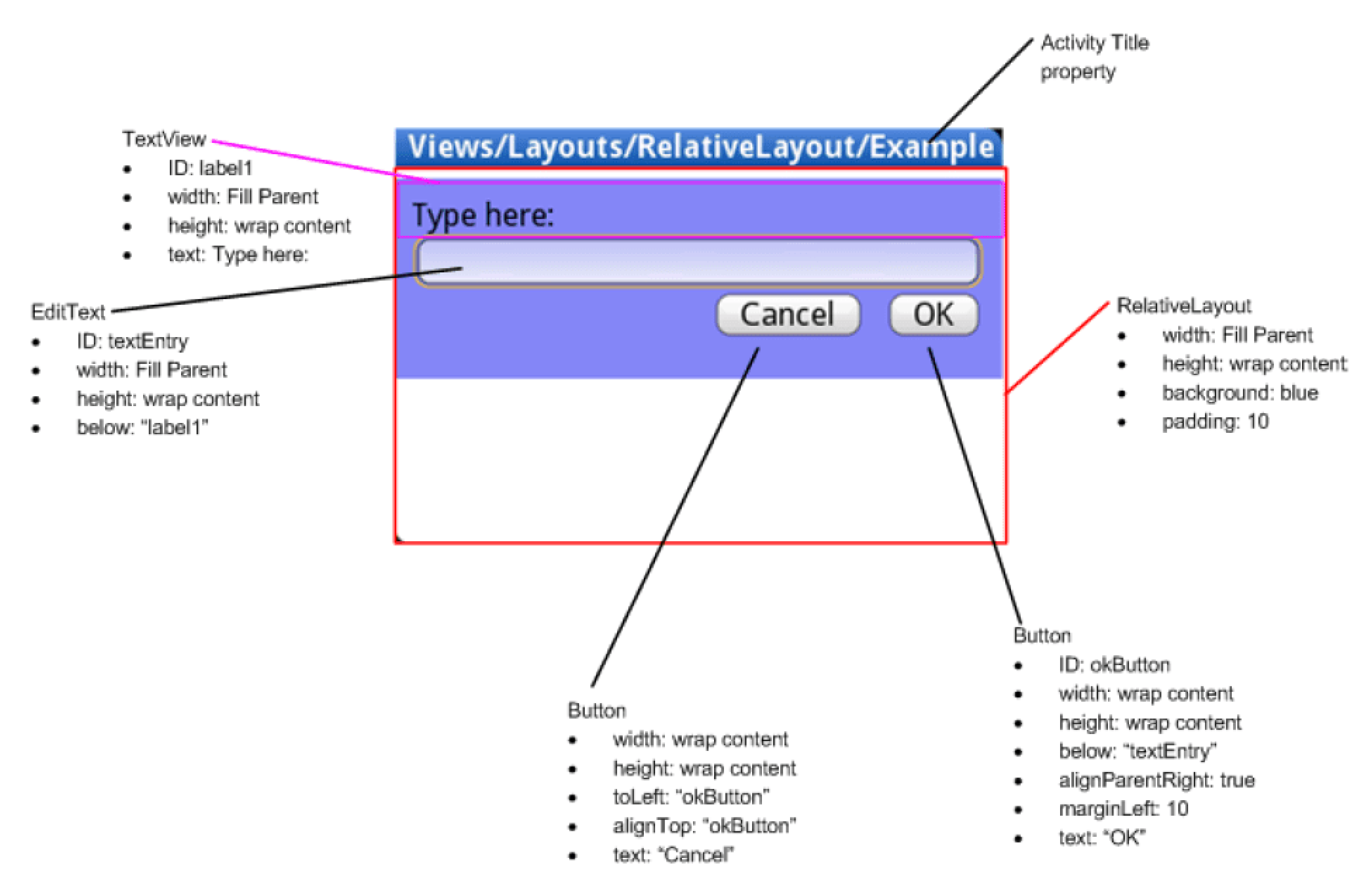


图 4-8　RelativeLayout 布局的结构

4.2.5　表格布局 TableLayout

表格布局 TableLayout 将子元素的位置分配到行或列中。一个 TableLayout 由许多的 TableRow 组成，每个 TableRow 都会定义一个 row。TableLayout 容器不会显示 row、columns 或 cell 的边框线。每个 row 拥有 0 个或多个 cell，每个 cell 拥有一个 View 对象。表格由列和行组成许多的单元格。表格允许单元格为空，单元格不能跨列。看下面的一段代码：

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout width="fill parent"
android:layout height="fill parent"
   android:stretchColumns="1">
   <TableRow>
      <TextView android:layout column="1" android:text="Open..." />
      <TextView android:text="Ctrl-O" android:gravity="right" />
   </TableRow>
   <TableRow>
      <TextView android:layout_column="1" android:text="Save..." />
      <TextView android:text="Ctrl-S" android:gravity="right" />
   </TableRow>
   //这里是上图中的分隔线
   <View android:layout height="2dip" android:background="#FF909090" />
   <TableRow>
      <TextView android:text="X" />
      <TextView android:text="Export..." />
      <TextView android:text="Ctrl-E" android:gravity="right " />
```

```
    </TableRow>
    <View android:layout_height="2dip" android:background="#FF909090" />
    <TableRow>
        <TextView android:layout_column="1" android:text="Quit"
            android:padding="3dip" />
    </TableRow>
</TableLayout>
```

上述代码使用了 TableLayout 布局方式，执行后的效果如图 4-9 所示。



图 4-9　TableLayout 布局

实例 2	
源码路径	\daima\4\RelativeLayout
功能	演示相对布局 RelativeLayout 的用法

本实例的具体实现流程如下。

(1) 在 Eclipse 中新建一个名为“RelativeLayout”的工程文件。

(2) 编写布局文件 main.xml，在里面分别插入 1 个 TextView 控件、1 个 EditText 控件、2 个 Button 控件。具体代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="请写上您的祝福:"/>
    <EditText
        android:id="@+id/entry"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@android:drawable/editbox_background"
        android:layout_below="@id/label"/>
    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/entry"
        android:layout_alignParentRight="true"
        android:layout_marginLeft="10dip"
```

```
        android:text="确定" />
    <Button
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout toLeftOf="@id/ok"
        android:layout alignTop="@id/ok"
        android:text="取消" />
</RelativeLayout>
```

至此，整个实例全部介绍完毕。执行后的效果如图 4-10 所示。



图 4-10　执行效果

注意： LayoutParams 参数的意义如下。

当把一个 View 加入到一个 Viewgroup 中后，例如加入到 RelativeLayout 里面，我们知道此时这个 View 在 RelativeLayout 里面是怎样显示的呢？答案其实很简单：当向里面加入 View 时，我们传递一组值，并将这组值封装在 LayoutParams 类中。这样当再显示这个 View 时，其容器会根据封装在 LayoutParams 的值来确认此 View 的显示大小和位置。由此可以看出，LayoutParams 的功能如下。

① 每一个 Viewgroup 类使用一个继承于 ViewGroup.LayoutParams 的嵌套类。

② 包含定义了子节点 View 的尺寸和位置的属性类型。

LayoutParams 的具体结构如图 4-11 所示。

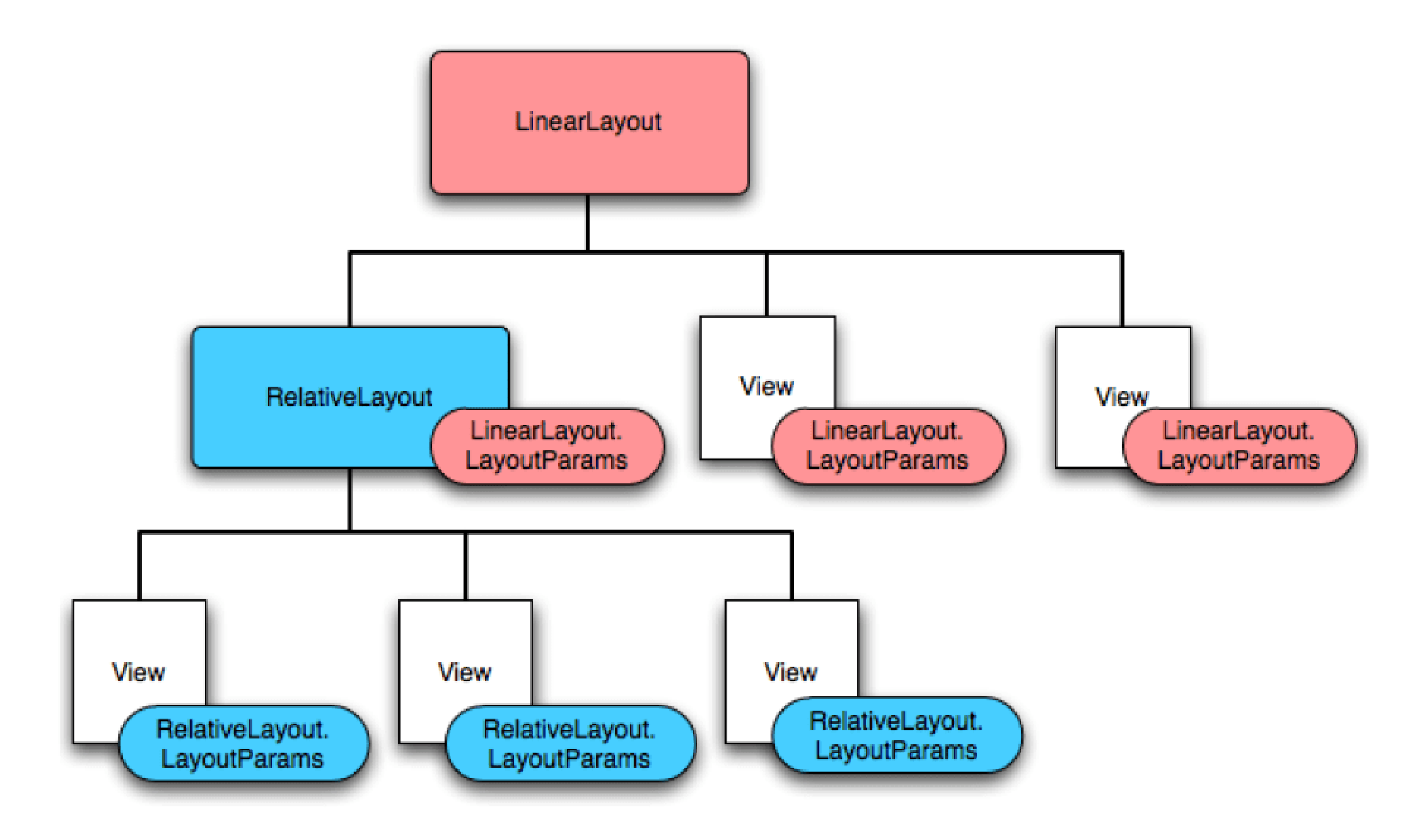


图 4-11　LayoutParams 的结构图

4.3 <merge/>标签在 UI 界面中的优化作用

在定义 Android Layout(XML)时，有 4 个比较特别的标签是非常重要的，其中有三个是与资源复用有关的。4 个标签分别是<viewStub/>、<requestFocus/>、<merge/>和<include/>。可是以往我们所接触的案例或者官方文档的例子都没有着重去介绍这些标签的重要性。其中<merge/>标签十分重要，因为它在优化 UI 结构时起到很重要的作用。<merge/>标签可以通过删减多余或者额外的层级，从而优化整个 Android Layout 的结构。

在使用<merge/>标签的时候需要注意以下两点。

(1) <merge/>只可以作为 xml layout 的根节点。

(2) 当需要扩充的 xml layout 本身是由 merge 作为根节点的话，需要将被导入的 xml layout 置于 viewGroup 中，同时需要设置 attachToRoot 为 True。

其实除了本例外，<merge/>标签还有另外一个用法。当应用 Include 或者 ViewStub 标签从外部导入 XML 结构时，可以将被导入的 XML 用 merge 作为根节点表示，这样当被嵌入父级结构中后可以很好地将它所包含的子集融合到父级结构中，而不会出现冗余的节点。

下面通过一个具体实例来说明<merge/>标签在 UI 界面中的优化作用。

实例 3	
源码路径	\daima\4\merge
功能	演示<merge/>标签

新建一个简单的 Layout 界面，在里面包含了两个 Views 元素，分别是 ImageView 和 TextView。在默认状态下将这两个元素放在 FrameLayout 中，效果是在主视图中全屏显示一张图片，之后将标题显示在图片上，并位于视图的下方。文件 main.xml 的主要实现代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="fill parent"
    android:layout height="fill parent"
    >
    <ImageView
        android:layout width="fill parent"
        android:layout height="fill parent"
        android:scaleType="center"
        android:src="@drawable/golden gate"
        />
    <TextView
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout marginBottom="20dip"
        android:layout_gravity="center_horizontal|bottom"
```

```
        android:padding="12dip"
        android:background="#AA000000"
        android:textColor="#ffffffff"
        android:text="Golden Gate"
        />
</FrameLayout>
```

此时执行后的效果如图 4-12 所示。

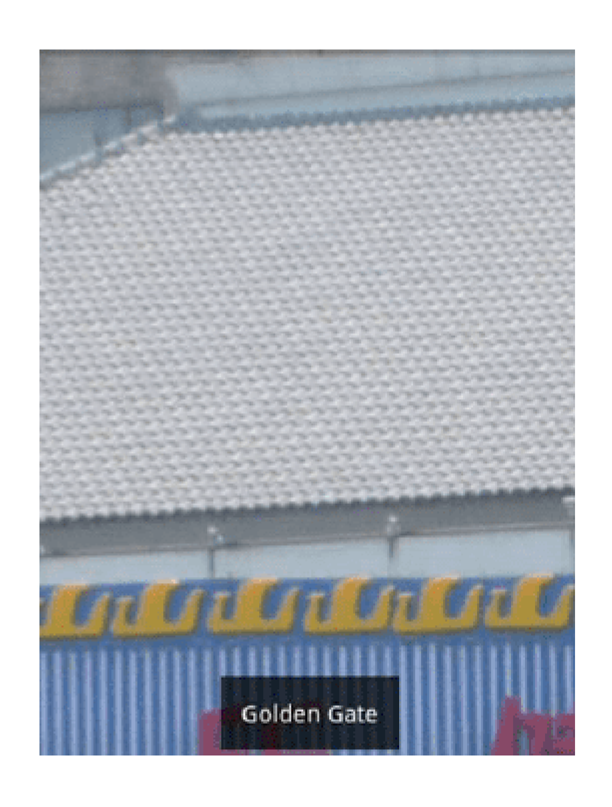


图 4-12　执行效果

启动 SDK 目录下的 tools 文件夹中的 hierarchyviewer.bat，如图 4-13 所示。

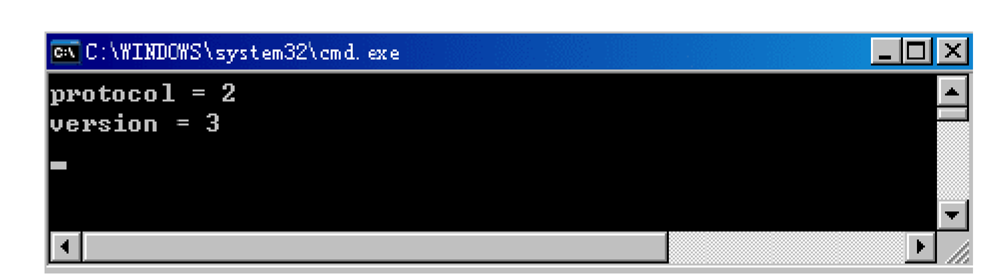


图 4-13　启动 hierarchyviewer.bat

此时可以查看当前 UI 的结构视图，如图 4-14 所示。

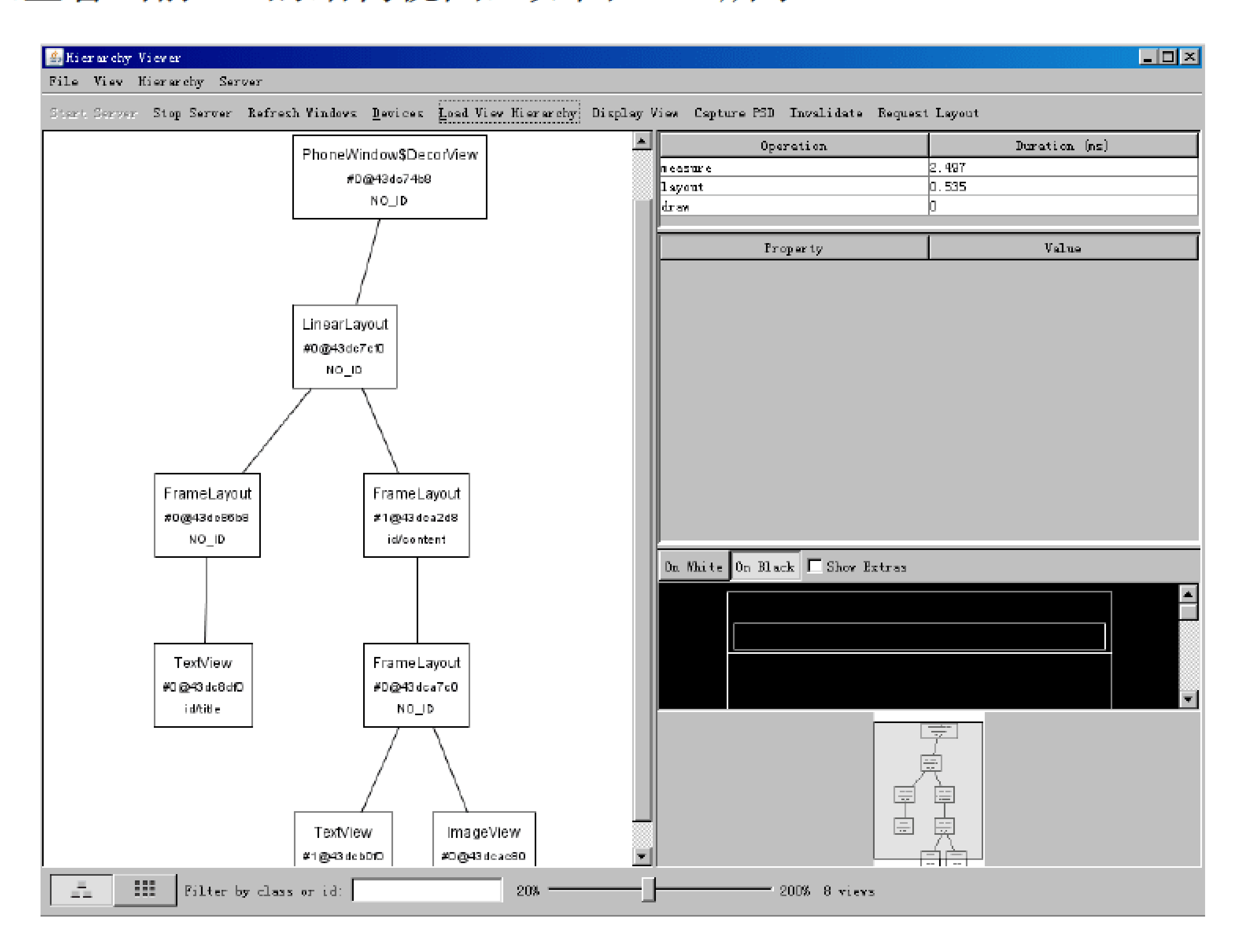


图 4-14　文件 main.xml 的 UI 结构视图

此时可以很明显地看到图中出现了两个 FrameLayout 节点，说明这两个完全意义相同的节点造成了资源浪费，那么如何才能解决呢？这时候就要用到<merge/>标签来处理类似的问题了。

将上面 XML 代码中的 FramLayout 换成 merge，文件 main2.xml 的具体实现代码如下。

```
<merge
    xmlns:android="http://schemas.android.com/apk/res/android"
    >
    <ImageView
        android:layout width="fill parent"
        android:layout height="fill parent"
        android:scaleType="center"
        android:src="@drawable/golden gate"
        />
    <TextView
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout marginBottom="20dip"
        android:layout gravity="center horizontal|bottom"
        android:padding="12dip"
        android:background="#AA000000"
        android:textColor="#ffffffff"
        android:text="Golden Gate"
        />
</merge>
```

此时程序运行后，在 Emulator 中显示的效果是一样的，可是通过 Hierarchy Viewer 查看的 UI 结构是有变化的，如图 4-15 所示。

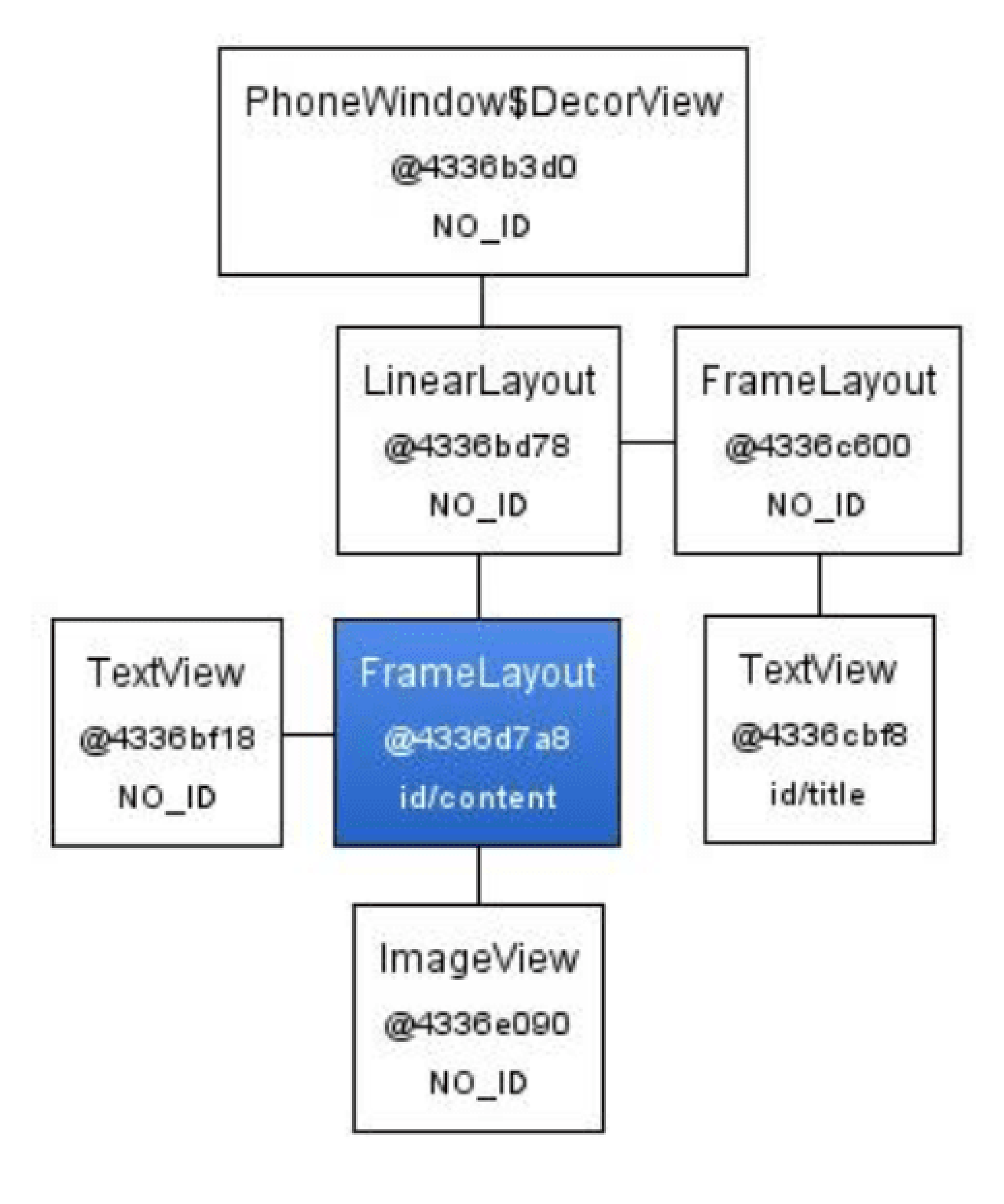


图 4-15　UI 结构视图

此时原来多余的 FrameLayout 节点被合并在一起了，即将<merge/>标签中的子集直接

加到 Activity 的 FrameLayout 根节点下。如果所创建的 Layout 并不是用 FramLayout 作为根节点(而是应用 LinerLayout 等定义 root 标签)，就不能应用上面的例子通过 merge 来优化 UI 结构。

4.4 遵循 Android Layout 优化的两段通用代码

Android 中的 Layout 优化一直是广大程序员们探讨的话题，接下来将给出两段通用的标准 XML 代码，并不是希望广大读者严格遵循下面的布局格式，而是希望根据自己项目的需求尽力向下面的标准靠拢。

第一段标注了 Layout 优化的 XML 代码：

```
<?xml version="1.0" encoding="utf-8"?>
<!--<FrameLayout-->
<!--  xmlns:android="http://schemas.android.com/apk/res/android"-->
<!--  android:layout width="fill parent"-->
<!--  android:layout height="fill parent">-->
<!--    <ListView android:id="@+id/list"-->
<!--           android:layout width="fill parent"-->
<!--           android:layout height="fill parent"/>-->
<!--    <TextView android:id="@+id/no item text"-->
<!--           android:layout width="fill parent"-->
<!--           android:layout height="fill parent"-->
<!--           android:gravity="center"-->
<!--           android:visibility="gone"/>-->
<!--</FrameLayout>-->
<merge xmlns:android="http://schemas.android.com/apk/res/android">
   <ListView android:id="@+id/list"
          android:layout width="fill parent"
          android:layout height="fill parent"/>
   <TextView android:id="@+id/no item text"
          android:layout width="fill parent"
          android:layout height="fill parent"
          android:gravity="center"
          android:visibility="gone"/>
</merge>
```

第二段标注了 Layout 优化的 XML 代码：

```
<?xml version="1.0" encoding="utf-8"?>
 <!--<LinearLayout-->
 <!--  xmlns:android="http://schemas.android.com/apk/res/android"-->
 <!--  android:orientation="vertical"-->
 <!--  android:layout width="fill parent"-->
 <!--  android:layout height="fill parent">-->
 <!--  -->
 <!--  <ImageView android:id="@+id/softicon"-->
 <!--            android:layout_width="wrap_content"-->
```

```
<!--            android:layout height="wrap content"-->
<!--            android:layout marginTop="10dip"-->
<!--            android:layout gravity="center"/>-->
  <TextView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/softname"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout marginBottom="10dip"
        android:layout gravity="center"
        android:gravity="center"
        android:drawableTop="@drawable/icon"/>
 <!--</LinearLayout>-->
```

在进行 Layout 布局时，必须注意下面的 4 点。

(1) 如果可能，尽量不要使用 LinearLayout，而是使用 RelativeLayout 替换它。这是因为 android:layout_alignWithParentIfMissing 只对 RelativeLayout 有用，如果那个视图设置为 gone，这个属性将按照父视图进行调整。

(2) 在使用 Adapter 控件时，例如 list，如果布局中递归太深，则会严重影响性能。

(3) 对于 TextView 和 ImageView 组成的 Layout 来说，可以直接使用 TextView 替换。

(4) 如果其父 Layout 是 FrameLayout，如果子 Layout 也是 FrameLayout，此时可以将 FrameLayout 替换为 merge，这样做的好处是可以减少层的递归深度。

4.5 优化 Bitmap 图片

4.5.1 实例说明

在 Android 项目中，如果直接使用 ImageView 显示 Bitmap 会占用较多资源。在图片较大的时候，甚至可能会导致系统崩溃。使用 BitmapFactory.Options 设置 inSampleSize，这样做可以减少对系统资源的要求。通过本实例，将演示优化 Android 程序中 Bitmap 图片的方法。

4.5.2 具体实现

(1) 编写文件 xml.xml，插入一个 ImageView 控件用于显示一幅图片，主要代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout width="fill parent"
android:layout height="fill parent"
>
<TextView
android:layout_width="fill_parent"
```

 UI 布局优化

```
android:layout height="wrap content"
android:text="@string/hello"
/>
<ImageView
android:id="@+id/imageview"
android:layout gravity="center"
android:layout width="fill parent"
android:layout height="fill parent"
android:scaleType="center"
/>
</LinearLayout>
```

(2) 编写文件 java.java，通过设置 inJustDecodeBounds 为 true 的方式来获取 outHeight(图片原始高度)和 outWidth(图片的原始宽度)，然后计算一个 inSampleSize(缩放值)。主要代码如下。

```
import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Bundle;
import android.widget.ImageView;
import android.widget.Toast;

public class AndroidImage extends Activity {

private String imageFile = "/sdcard/AndroidSharedPreferencesEditor.png";
/** Called when the activity is first created. */

@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);

ImageView myImageView = (ImageView)findViewById(R.id.imageview);
//Bitmap bitmap = BitmapFactory.decodeFile(imageFile);
//myImageView.setImageBitmap(bitmap);

Bitmap bitmap;
float imagew = 300;
float imageh = 300;

BitmapFactory.Options bitmapFactoryOptions = new BitmapFactory.Options();
bitmapFactoryOptions.inJustDecodeBounds = true;
bitmap = BitmapFactory.decodeFile(imageFile, bitmapFactoryOptions);

int yRatio = (int)Math.ceil(bitmapFactoryOptions.outHeight/imageh);
int xRatio = (int)Math.ceil(bitmapFactoryOptions.outWidth/imagew);

if (yRatio > 1 || xRatio > 1){
```

```
 if (yRatio > xRatio) {
  bitmapFactoryOptions.inSampleSize = yRatio;
  Toast.makeText(this,
    "yRatio = " + String.valueOf(yRatio),
    Toast.LENGTH_LONG).show();
 }
 else {
  bitmapFactoryOptions.inSampleSize = xRatio;
  Toast.makeText(this,
    "xRatio = " + String.valueOf(xRatio),
    Toast.LENGTH LONG).show();
 }
}
else{
 Toast.makeText(this,
   "inSampleSize = 1",
   Toast.LENGTH LONG).show();
}
bitmapFactoryOptions.inJustDecodeBounds = false;
bitmap = BitmapFactory.decodeFile(imageFile, bitmapFactoryOptions);
myImageView.setImageBitmap(bitmap);
}
}
```

在上述代码中，属性 inSampleSize 表示缩略图大小为原始图片大小的几分之一，即如果这个值为 2，则取出的缩略图的宽和高都是原始图片的 1/2，图片大小就为原始大小的 1/4。

Options 中的属性 inJustDecodeBounds 比较重要，如果设置 inJustDecodeBounds 为 true，则可以获取 outHeight(图片原始高度)和 outWidth(图片的原始宽度)的值，通过这两个值就可以计算对应的 inSampleSize(缩放值)。

4.6 FrameLayout 布局优化

经过本章前面的内容可知，我们可以把 FrameLayout 当作 canvas(画布)，固定从屏幕的左上角开始填充图片和文字等。例如下面的演示代码，原来可以利用 android:layout_gravity 来设置。

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout width="fill parent"
  android:layout height="fill parent" >

   <ImageView
      android:id="@+id/image"
```

 UI 布局优化

```
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:scaleType="center"
        android:src="@drawable/candle"
        />
    <TextView
        android:id="@+id/text1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:textColor="#00ff00"
        android:text="@string/hello"
        />
    <Button
        android:id="@+id/start"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom"
        android:text="Start"
        />
</FrameLayout>
```

执行上述代码后，效果如图 4-16 所示。

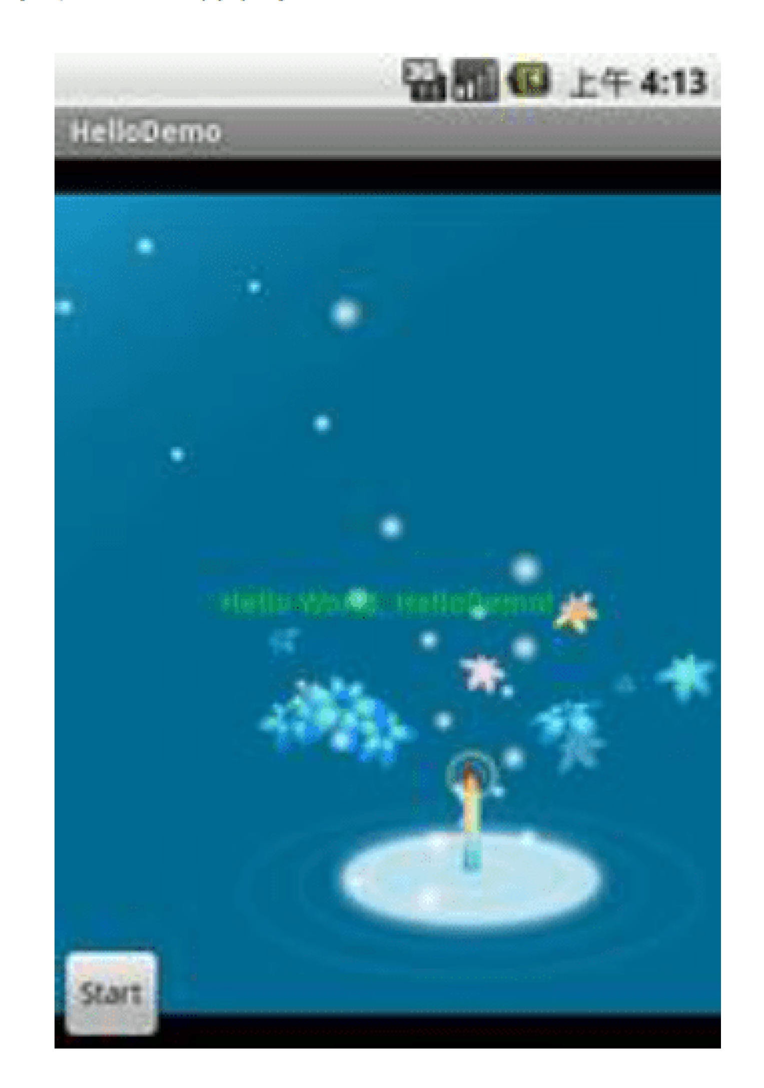


图 4-16　执行效果

使用 tools 里面的 hierarchyviewer.bat 来查看 Layout 的层次。在模拟器中启动所要分析的程序，再启动 hierarchyviewer.bat，查看到的 UI 的结构视图如图 4-17 所示。

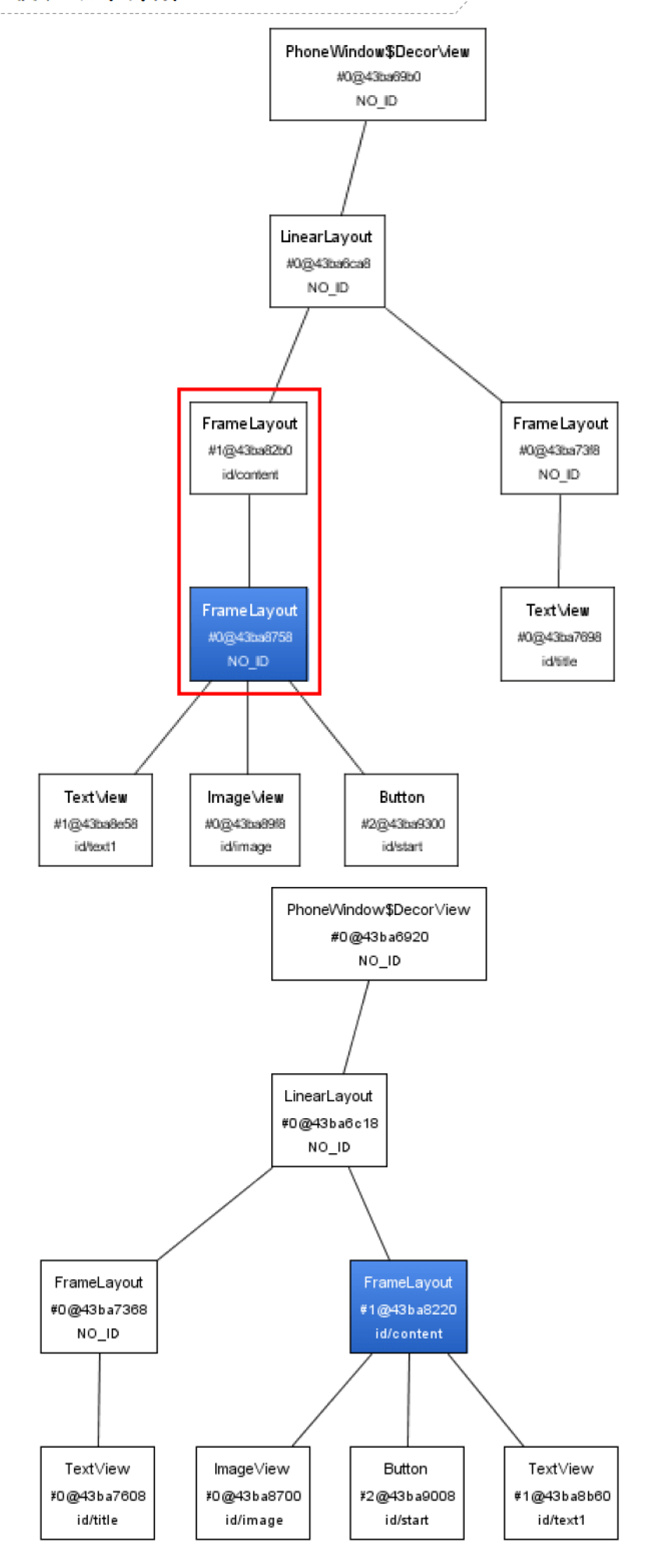


图 4-17　UI 的结构视图

 UI 布局优化

4.6.1 使用<merge>减少视图层级结构

从图 4-17 中可以看到存在两个 FrameLayout(粗线框中的两个)。如果能在 Layout 文件中把 FrameLayout 声明去掉就可以进一步优化布局代码了。但是由于布局代码需要外层容器容纳，如果直接删除 FrameLayout，则该文件就不是合法的布局文件。这种情况下就可以使用<merge> 标签了。我们可以对代码进行如下修改即可消除多余的 FrameLayout。

```
<?xml version="1.0" encoding="utf-8"?>
<merge  xmlns:android="http://schemas.android.com/apk/res/android">
    <ImageView
        android:id="@+id/image"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:scaleType="center"
        android:src="@drawable/candle"
        />
    <TextView
        android:id="@+id/text1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:textColor="#00ff00"
        android:text="@string/hello"
        />
    <Button
        android:id="@+id/start"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom"
        android:text="Start"
        />
</merge>
```

虽然这样可以减少视图层级结构，实现了对 UI 的优化，但是<merge>也有一些使用限制，例如只能用于 XML Layout 文件的根元素；在代码中使用 LayoutInflater.Inflater()一个以 merge 为根元素的布局文件的时候，需要使用 View inflate(int resource,ViewGroup root, boolean attachToRoot)指定一个 ViewGroup 作为其容器，并且要设置 attachToRoot 为 true。

4.6.2 使用<include>重用 Layout 代码

Android 平台提供了大量的 UI 构件，你可以将这些小的视觉块(构件)搭建在一起，呈现给用户复杂且有用的画面。然而，应用程序有时需要一些高级的视觉组件。为了满足这一需求，并且能高效地实现，你可以把多个标准的构件结合起来成为一个单独的、可重用的组件。

和常见的程序开发一样，我们可以使用<include>来包含重用的 Layout 代码。假如在某

个布局里面需要用到另一个相同的布局设计，我们就可以通过<include>标签来重用 Layout 代码：

```
    <?xml version="1.0" encoding="utf-8"?>
    <LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout width="fill parent"
        android:layout height="fill parent">
        <include android:id="@+id/layout1" layout="@layout/relative" />
        <include android:id="@+id/layout2" layout="@layout/relative" />
        <include android:id="@+id/layout3" layout="@layout/relative" />
    </LinearLayout>
```

在这里需要注意的是，“@layout/relative”不是引用 Layout 的 id，而是引用 res/layout/relative.xml，其内容可以是随意设置的布局代码，例如可以是下面的代码。

```
    <?xml version="1.0" encoding="utf-8"?>
    <RelativeLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout width="wrap content"
      android:layout height="wrap content"
      android:id="@+id/relativelayout">

        <ImageView
            android:id="@+id/image"
            android:layout width="wrap content"
            android:layout height="wrap content"
            android:src="@drawable/icon"
            />
        <TextView
            android:id="@+id/text1"
            android:layout width="fill parent"
            android:layout height="wrap content"
            android:text="@string/hello"
            android:layout toRightOf="@id/image"
            />
        <Button
            android:id="@+id/button1"
            android:layout width="fill parent"
            android:layout height="wrap content"
            android:text="button1"
            android:layout toRightOf="@id/image"
            android:layout below="@id/text1"
            />
    </RelativeLayout>
```

执行后的效果如图 4-18 所示。

另外，使用<include>标签以后，除了可以覆写 id 属性值外，还可以修改其他属性值，例如 android:layout_width 和 android:height 等。

 UI 布局优化

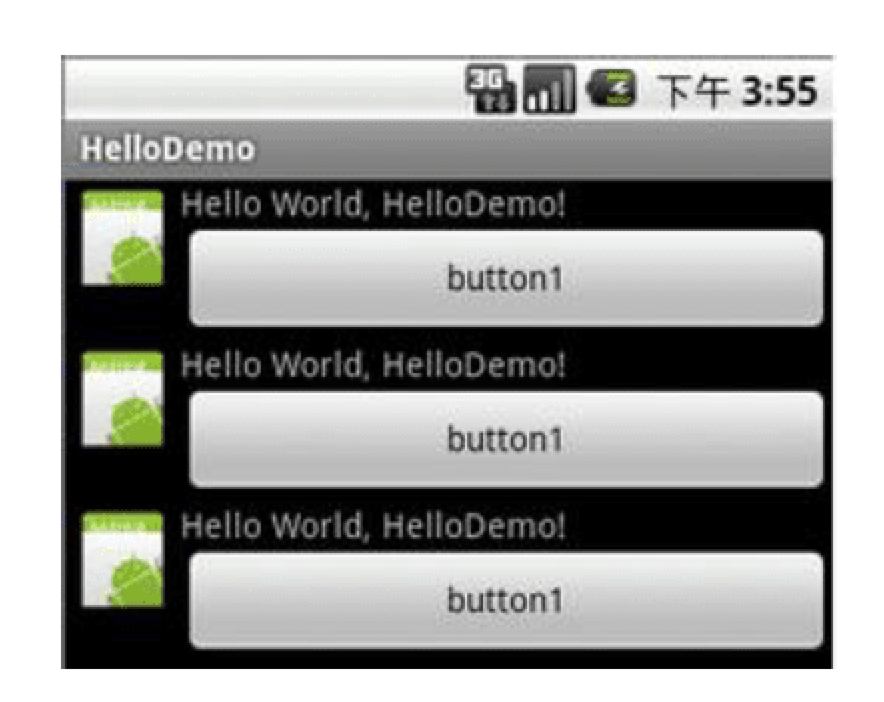


图 4-18　执行效果

你可以创建一个可重用的组件包含一个进度条和一个取消按钮，一个 Panel 包含两个按钮(确定和取消动作)，一个 Panel 包含图标、标题和描述，等等。简单的，你可以通过书写一个自定义的 View 来创建一个 UI 组件，但更简单的方式是仅使用 XML 来实现。

在 Android XML 布局文件里，通常每个标签都对应一个真实的类实例(这些类一般都是 View 的子类)。UI 工具包还允许你使用三个特殊的标签，它们不对应具体的 View 实例：<requestFocus/>、<merge/>、<include/>。

由此可见，<include/>元素的作用如同它的名字一样，用于包含其他的 XML 布局。再看下面使用 include 标签的例子：

```
<com.android.launcher.Workspace
   android:id="@+id/workspace"
   android:layout width="fill parent"
   android:layout height="fill parent"
   launcher:defaultScreen="1">
   <include android:id="@+id/cell1" layout="@layout/workspace screen" />
   <include android:id="@+id/cell2" layout="@layout/workspace screen" />
   <include android:id="@+id/cell3" layout="@layout/workspace screen" />
</com.android.launcher.Workspace>
```

在上述<include/>代码中，只需要 Layout 特性。此特性不带 Android 命名空间前缀，这表示我们想包含的布局的引用。在上述例子中，相同的布局被包含了三次。这个标签还允许你重写被包含布局的一些特性。上面的例子显示了你可以使用 android:id 来指定被包含布局中根 View 的 id；它还可以覆盖已经定义的布局 id。同样道理，我们可以重写所有的布局参数。这意味着任何 android:layout_*的特性都可以在<include/>中使用。例如下面的代码：

```
<include android:layout width="fill parent"
layout="@layout/image holder" />
<include android:layout_width="256dip" layout="@layout/image_holder" />
```

标签<include/>非常重要，特别是在依据设备定制 UI 的时候表现得尤为有用。例如 Activity 的主要布局放置在“layout/”文件夹下，其他布局放置在“layout-land/”和“layout-port/”下。这样，在垂直和水平方向时你可以共享大多数的 UI 布局。

4.6.3 延迟加载

延迟加载的功能非常重要，特别是在界面中显示的内容比较多并且所占空间比较大时。在 Android 应用程序中，可以使用 ViewStub 实现延迟加载功能。ViewStub 是一个不可见的、大小为 0 的 View(视图)，最佳用途就是实现 View 的延迟加载，在需要的时候再加载 View，这和 Java 中常见的性能优化方法延迟加载一样。

当调用 ViewStub 的 setVisibility()函数设置为可见或调用 inflate 初始化该 View 的时候，ViewStub 引用的资源开始初始化，然后引用的资源替代 ViewStub 自己的位置填充在 ViewStub 的位置。在没有调用 setVisibility(int)函数或 inflate()函数之前，ViewStub 一直存在于组件树层级结构中。但是由于 ViewStub 非常轻量级，所以对性能影响非常小。可以通过 ViewStub 的 inflatedId 属性来重新定义引用的 layout id。 例如下面的代码：

```
<ViewStub android:id="@+id/stub"
        android:inflatedId="@+id/subTree"
        android:layout="@layout/mySubTree"
        android:layout_width="120dip"
        android:layout_height="40dip" />
```

在上述代码中定义了 ViewStub，这可以通过 id "stub"来找到。在初始化资源 mySubTree 后，从父组件中删除了 stub，然后用“mySubTree”替代了 stub 的位置。初始资源 mySubTree 得到的组件可以通过 inflatedId 指定的 id:"subTree"来引用。最后初始化后的资源被填充到一个宽为 120dip、高为 40dip 的位置。

在初始化 ViewStub 对象时，建议读者使用下面的方式来实现。

```
ViewStub stub = (ViewStub) findViewById(R.id.stub);
View inflated = stub.inflate();
```

当调用函数 inflate()的时候，ViewStub 被引用的资源替代，并且返回引用的 View。这样程序可以直接得到引用的 View，而无须再次调用函数 findViewById()来查找了，这样提高了效率，达到了优化的目的。

注意： ViewStub 优化方式也不是万能的，其中最大的缺陷是暂时还不支持<merge />标签。

4.7 使用 Android 为我们提供的优化工具

考虑到优化的重要性，所以 Android 为我们提供了专业的优化工具，这些工具都包含在 Android SDK 包中。在本节的内容中，将详细讲解这些优化工具的基本用法。

4.7.1 Layout Optimization 工具

通过 Layout Optimization 工具可以分析所提供的 Layout，并提供优化意见。读者可以

在 tools 文件夹中找到 layoutopt.bat 并启动。

接下来再介绍另一个布局优化工具——layoutopt。这是 Android 为我们提供的布局分析工具。它能分析指定的布局，然后提出优化建议。

要想运行它，需要打开命令行进入 SDK 的 tools 目录，输入 Layoutopt 加上我们的布局目录命令行。运行后如图 4-19 所示，其中框出的部分即为该工具分析布局后提出的建议，这里为建议替换标签。

```
E:\TDDOWNLOAD\android-sdk_r06-windows\android-sdk-windows\tools>layoutopt E:\wor
kspace\EX_03_07\res\layout\main.xml
E:\workspace\EX_03_07\res\layout\main.xml
        6:37 The root-level <FrameLayout/> can be replaced with <merge/>
E:\TDDOWNLOAD\android-sdk_r06-windows\android-sdk-windows\tools>
```

图 4-19 命令行

由此可见，通过这个工具，能很好地优化我们的 UI 设计，寻找到更好的布局方法。Layout Optimization 工具的用法如下：

```
layoutopt  <list of xml files or directories>
```

其参数是一个或多个 Layout XML 文件，以空格间隔。也可以是多个 Layout XML 文件所在的文件夹路径。例如下面演示了 Layout Optimization 工具的用法：

```
layoutopt  G:\StudyAndroid\UIDemo\res\layout\main.xml
layoutopt  G:\StudyAndroid\UIDemo\res\layout\main.xml
G:\StudyAndroid\UIDemo\res\layout\relative.xml
layoutopt  G:\StudyAndroid\UIDemo\res\layout
```

其实 UI 优化是需要一定技巧的，性能良好的代码固然重要，但写出优秀代码的成本往往也很高。很多读者可能不会过早地贸然为那些只运行一次或临时功能代码实施优化，如果你的应用程序反应迟钝，并且卖得很贵，或使系统中的其他应用程序变慢，用户一定会有所响应，你的应用程序下载量将很可能受到影响。

为了节省成本，在开发期间我们应该尽早优化布局。通过使用 Android SDK 提供的工具 Layout Optimization，可以自动分析我们的布局，发现可能并不需要的布局元素，以降低布局复杂度。在接下来的内容中，将通过一个具体演示来说明使用 Layout Optimization 的基本流程。

(1) 准备工作

如果想使用 Android SDK 中提供的优化工具，则需要在开发系统的命令行中工作，如果不熟悉使用命令行工具，那么你得多下功夫学习了。笔者在此强烈建议将 Android 工具所在的路径添加到操作系统的环境变量中，这样就可以直接敲名字运行相关的工具了，否则每次都要在命令提示符后面输入完整的文件路径。假设在 Android SDK 中有两个工具目录：/tools 和/platform-tools，下面的演示将主要使用位于/tools 目录中的 Layoutopt 工具，另外我想说的是，ADB 工具位于/platform-tools 目录下。

(2) 运行 Layoutopt

运行 Layoutopt 工具的方法相当简单，只需要跟上一个布局文件或布局文件所在目录作为参数。在此需要注意的是，这里必须包括布局文件或目录的完整路径，即使当前就位于这个目录。请读者看一个简单的例子：

```
D:\d\tools\eclipse\article_ws\Nothing\res\layout>layoutopt
D:\d\tools\eclipse\article_ws\Nothing\res\layout\main.xml
D:\d\tools\eclipse\article_ws\Nothing\res\layout\main.xml
D:\d\tools\eclipse\article_ws\Nothing\res\layout>
```

在上述演示示例中，包含了文件的完整路径，如果不指定完整路径，不会输出任何内容，例如：

```
D:\d\tools\eclipse\article_ws\Nothing\res\layout>layoutopt main.xml
D:\d\tools\eclipse\article_ws\Nothing\res\layout>
```

如果读者看不到任何东西，则很可能是文件未被解析的原因，也就是说文件可能未被找到。

(3) 使用 Layoutopt 输出

Layoutopt 的输出结果只是概括性的建议，我们可以有选择地在应用程序中采纳这些建议，下面来看几个使用 layoutopt 输出建议的例子。

① 建议 1：无用的布局

在布局设计期间通常会频繁地移动各种组件，并且有些组件最终可能会不再使用，例如下面的布局代码：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
android:orientation="horizontal">
<LinearLayout        android:id="@+id/linearLayout1"
 android:layout_height="wrap_content"
 android:layout_width="wrap_content"
 android:orientation="vertical">
 <TextView           android:id="@+id/textView1"
 android:layout_width="wrap_content"
 android:text="TextView"
android:layout_height="wrap_content"></TextView>
</LinearLayout>
</LinearLayout>
```

Layout Optimization 工具将会很快输出如下提示，告诉我们 LinearLayout 内的 LinearLayout 是多余的。

```
11:17 This LinearLayout layout or its LinearLayout parent is useless
```

在上述输出结果中，每一行最前面的两个数字表示建议的行号。

② 建议 2：根可以替换

Layoutopt 的输出有时是矛盾的，例如下面的布局代码：

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout width="match parent"
android:layout height="match parent">
<LinearLayout        android:id="@+id/linearLayout1"
android:layout height="wrap content"
android:layout width="wrap content"
android:orientation="vertical">
<TextView            android:id="@+id/textView1"
android:layout width="wrap content"
android:text="TextView"
android:layout height="wrap content"></TextView>
<TextView            android:text="TextView"
android:id="@+id/textView2"
android:layout width="wrap content"
android:layout height="wrap content"></TextView>
</LinearLayout>
</FrameLayout>
```

Layout Optimization 工具将会返回下面的输出：

```
5:22 The root-level <FrameLayout/> can be replaced with <merge/>
10:21 This LinearLayout layout or its FrameLayout parent is useless
```

其中第一行的建议虽然可行，但不是必需的，我们希望两个 TextView 垂直放置，因此 LinearLayout 应该保留，而第二行的建议则可以采纳，可以删除无用的 FrameLayout。但是这个工具不是全能的，例如在上面的演示代码中，如果给 FrameLayout 添加一个背景属性，然后再运行工具，第一个建议会消失，而第二个建议仍然会显示。工具 Layout Optimization 知道我们不能通过合并控制背景，但检查了 LinearLayout 后，它似乎就忘了还给 FrameLayout 添加了一个 LinearLayout 不能提供的属性。

③ 建议 3：太多的视图

其实每个视图都会消耗内存，如果在一个布局中布置太多的视图，布局会占用过多的内存。假设一个布局包含超过 80 个视图，则 Layout Optimization 可能会给出下面这样的建议：

```
-1:-1 This layout has too many views: 83 views, it should have <= 80!  -
1:-1 This layout has too many views: 82 views, it should have <= 80!  -
1:-1 This layout has too many views: 81 views, it should have <= 80!
```

上面的建议提示视图数量不能超过 80，当然最新的设备有可能支持这么多视图，但如果真的出现性能不佳的情况，建议最好采纳这个建议。

④ 建议 4：嵌套太多

在一个布局中不应该有太多的嵌套，Android 开发团队建议布局保持在 10 级以内，即使是最大的平板电脑屏幕，布局也不应该超过 10 级。当布局嵌套太多时，Layout Optimization 会输出如下内容：

```
-1:-1 This layout has too many nested layouts: 12 levels, it should have
<= 10!  305:318 This LinearLayout layout or its RelativeLayout parent is
```

```
possibly useless  307:314 This LinearLayout layout or its FrameLayout
parent is possibly useless  310:312 This LinearLayout layout or its
LinearLayout parent is possibly useless
```

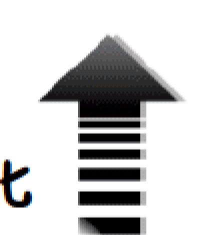
上述内容表示嵌套布局通常伴随有一些无用布局的警告，有助于找出哪些布局可以移除，避免屏幕布局全部重新设计。

由此可见，Layout Optimization 是一个快速易用的布局分析工具，找出低效和无用的布局，你要做的是判断是否采纳 Layoutopt 给出的优化建议，虽然采纳建议做出修改不会立即大幅改善性能，但没有理由需要复杂的布局拖慢整个应用程序的速度，并且后期的维护难度也很大。简单布局不仅简化了开发周期，还可以减少测试和维护的工作量，因此，在应用程序开发期间，应尽早优化你的布局，不要等到最后用户反馈回来再做修改。

4.7.2 Hierarchy Viewer 工具

层级观察器 Hierarchy Viewer 是 Android 为我们提供的一个优化工具，该工具是一个非常好的布局优化工具，可以实现 UI 优化功能。其实在前面的 4.3 节中已经使用过这个工具。为了进一步说明 Hierarchy Viewer 工具的用法，请看下面的一段 UI 代码：

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:Android="http://schemas.android.com/apk/res/android"
   Android:orientation="vertical"
   Android:layout width="fill parent"
   Android:layout height="fill parent"
   >
<TextView
   Android:layout width="300dip"
   Android:layout height="300dip"
   Android:background="#00008B"
   Android:layout gravity="center"
   />
<TextView
   Android:layout width="250dip"
   Android:layout height="250dip"
   Android:background="#0000CD"
   Android:layout gravity="center"
   />
<TextView
   Android:layout_width="200dip"
   Android:layout height="200dip"
   Android:background="#0000FF"
   Android:layout gravity="center"
   />
<TextView
   Android:layout width="150dip"
   Android:layout height="150dip"
   Android:background="#00BFFF"
   Android:layout gravity="center"
   />
<TextView
```



```
    Android:layout width="100dip"
    Android:layout height="100dip"
    Android:background="#00CED1"
    Android:layout gravity="center"
    />
</FrameLayout>
```

这是非常简单的一个布局界面，执行后可以实现如图 4-20 所示的层叠效果。

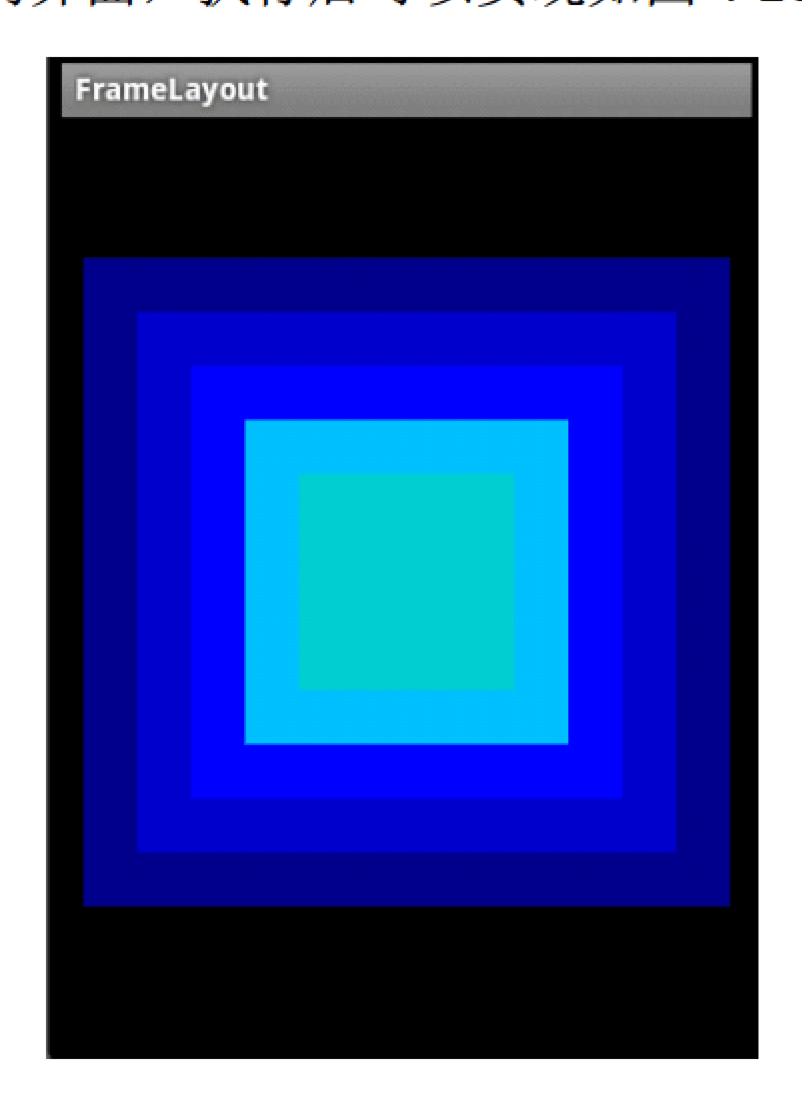


图 4-20　层叠效果

下面就用层级观察器 Hierarchy Viewer 来观察我们的布局，此工具在 SDK 的 tools 目录下，打开后的界面如图 4-21 所示。

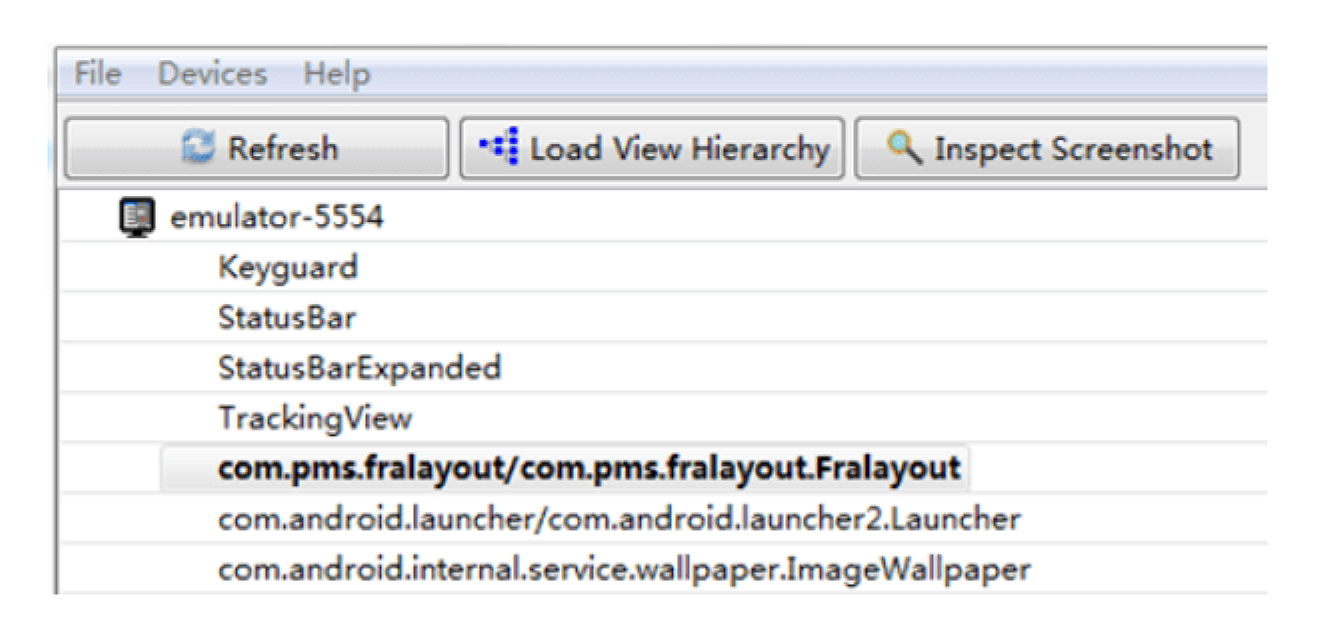


图 4-21　Hierarchy Viewer 界面

由此可见，Hierarchy Viewer 的界面很简洁，在里面列出了当前设备上的进程，并且在前台的进程加粗显示。上面有三个选项，分别是刷新进程列表，将层次结构载入到树形图，截取屏幕到一个拥有像素栅格的放大镜中。对应的在左下角可以进行三个视图的切换。在模拟器上打开写好的框架布局，在页面上选择，单击 Load View，进入如图 4-22 所示的界面。

其中左边的大图为应用布局的树形结构，上面写有控件名称和 id 等信息，下方的圆形表示这个节点的渲染速度，从左至右分别为测量大小、布局和绘制。绿色最快，红色最慢。右下角的数字为子节点在父节点中的索引，如果没有子节点则为 0。单击可以查看对应控件预览图、该节点的子节点数(为 6 则有 5 个子节点)以及具体渲染时间。双击可以打

开控件图。右侧是树形结构的预览、控件属性和应用界面的结构预览。单击相应的树形图中的控件可以在右侧看到它在布局中的位置和属性。工具栏中有一系列的工具，保存为png、psd或刷新等工具。其中有一个Load Overlay选项可以加入新的图层。当需要在你的布局中放上一个bitmap，可以用它来帮助你布局。单击左下角的第三个图标切换到像素视图，如图4-23所示。

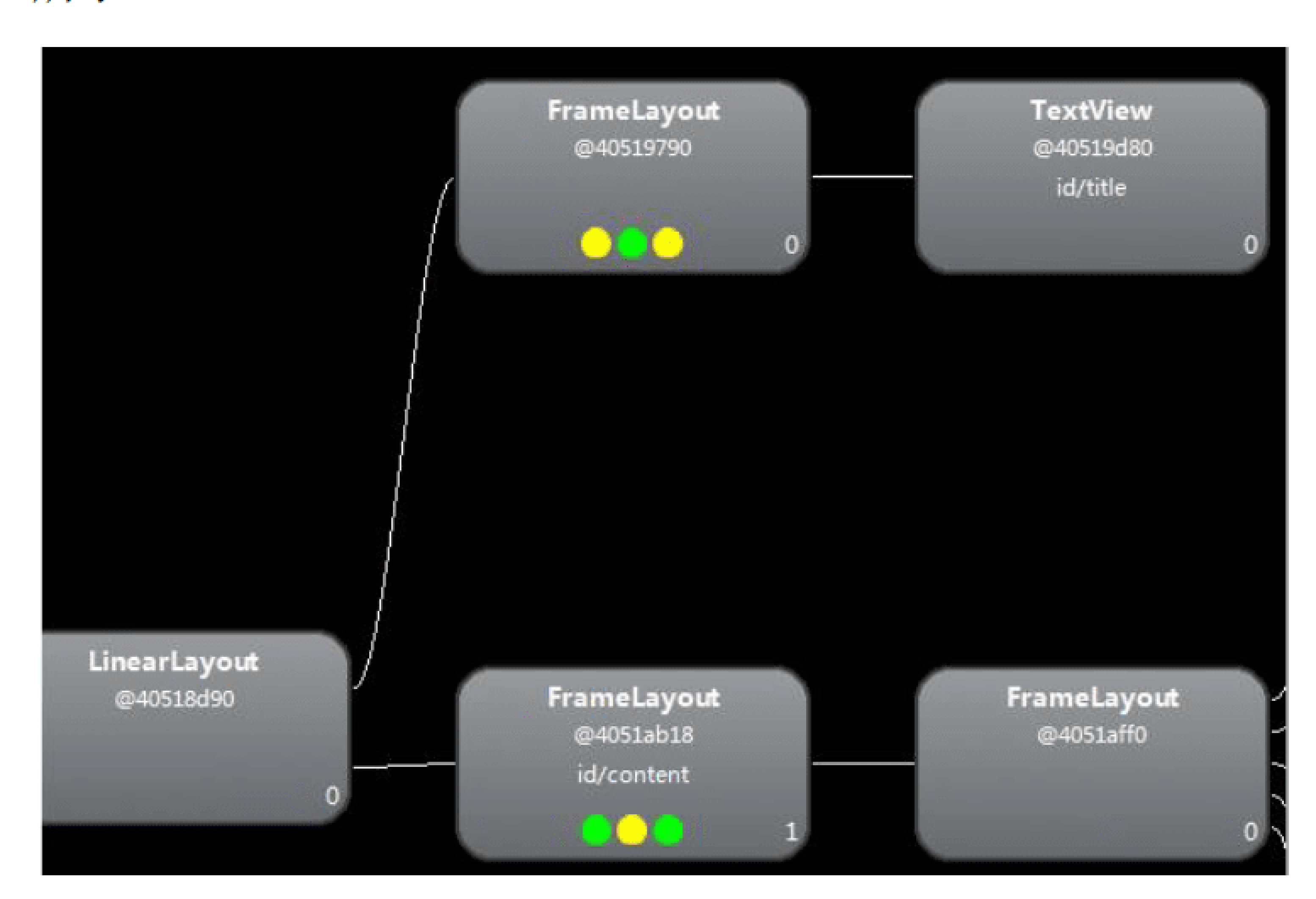


图 4-22　单击 Load View 后的界面

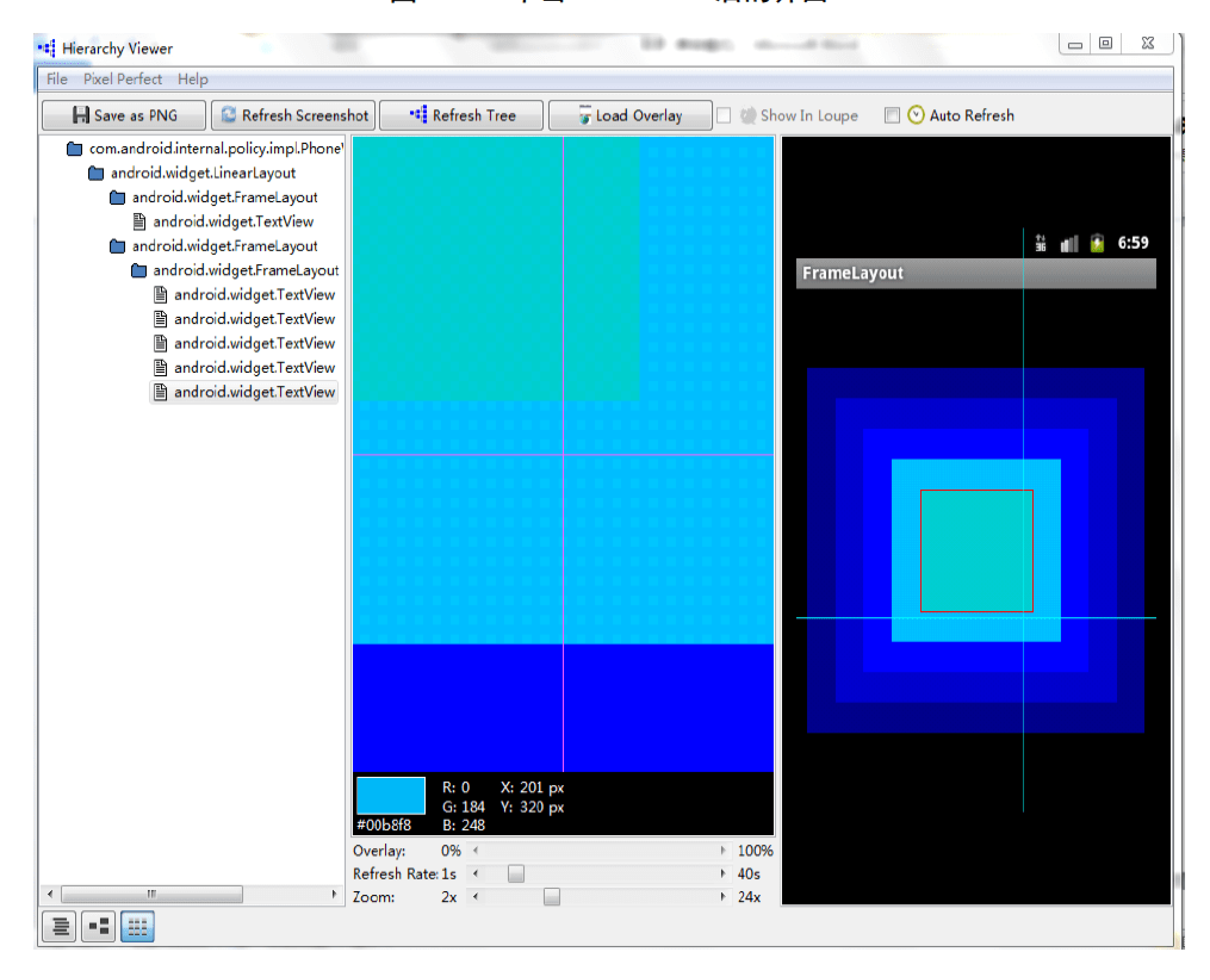


图 4-23　像素视图

 UI 布局优化



在上述视图左侧为 View 和 ViewGroup 关系图，单击其中的 View 会在右边的界面中用红色线条为我们选中相应的 View。最右侧为设备上的原图。中间为放大后带像素栅格的图像，可以在 Zoom 栏调整放大倍数。在这里能定位控件的坐标、颜色。观察布局就更加方便了。

4.7.3 联合使用<merge/>和<include/>标签实现互补

在接下来的内容中，将向读者介绍<merge/>标签和<include/>标签的互补使用。<merge/>标签用于减少 View 树的层次来优化 Android 的布局。通过下面的演示代码，就会很容易理解这个标签能解决的问题。下面的 XML 布局代码显示一幅图片，并且有一个标题位于其上方。这个结构相当简单，FrameLayout 里放置了一个 ImageView，其上放置了一个 TextView。

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <ImageView
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:scaleType="center"
      android:src="@drawable/golden_gate" />
   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginBottom="20dip"
      android:layout_gravity="center_horizontal|bottom"
      android:padding="12dip"
      android:background="#AA000000"
      android:textColor="#ffffffff"
      android:text="Golden Gate" />
</FrameLayout>
```

整段代码的布局渲染起来很漂亮，效果如图 4-24 所示。当使用 Hierarchy Viewer 工具来检查时，会发现事情变得很有趣。如果你仔细查看 View 树，将会注意到在 XML 文件中定义的 FrameLayout(高亮显示)是另一个 FrameLayout 唯一的子元素，如图 4-25 所示。

既然 FrameLayout 和它的父元素有着相同的尺寸(归功于 fill_parent 常量)，并且也没有定义任何的 background(背景)和额外的 padding(边缘)，所以它完全是无用的。我们所要做的仅仅是让 UI 变得更为复杂而已。我们怎样才能摆脱这个 FrameLayout 呢？毕竟，XML 文档需要一个根标签且 XML 布局总是与相应的 View 实例相对应，这时候就需要<merge/>标签来实现。当 LayoutInflater 遇到<merge/>标签时会跳过它，并将<merge/>内的元素添加到<merge/>的父元素里。下面我们将用<merge/>来替换 FrameLayout，并重写之前的 XML 布局。

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">
   <ImageView
      android:layout_width="fill_parent"
```

```
        android:layout height="fill parent"
        android:scaleType="center"
        android:src="@drawable/golden gate" />
    <TextView
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:layout marginBottom="20dip"
        android:layout gravity="center horizontal|bottom"
        android:padding="12dip"
        android:background="#AA000000"
        android:textColor="#ffffffff"
        android:text="Golden Gate" />
</merge>
```

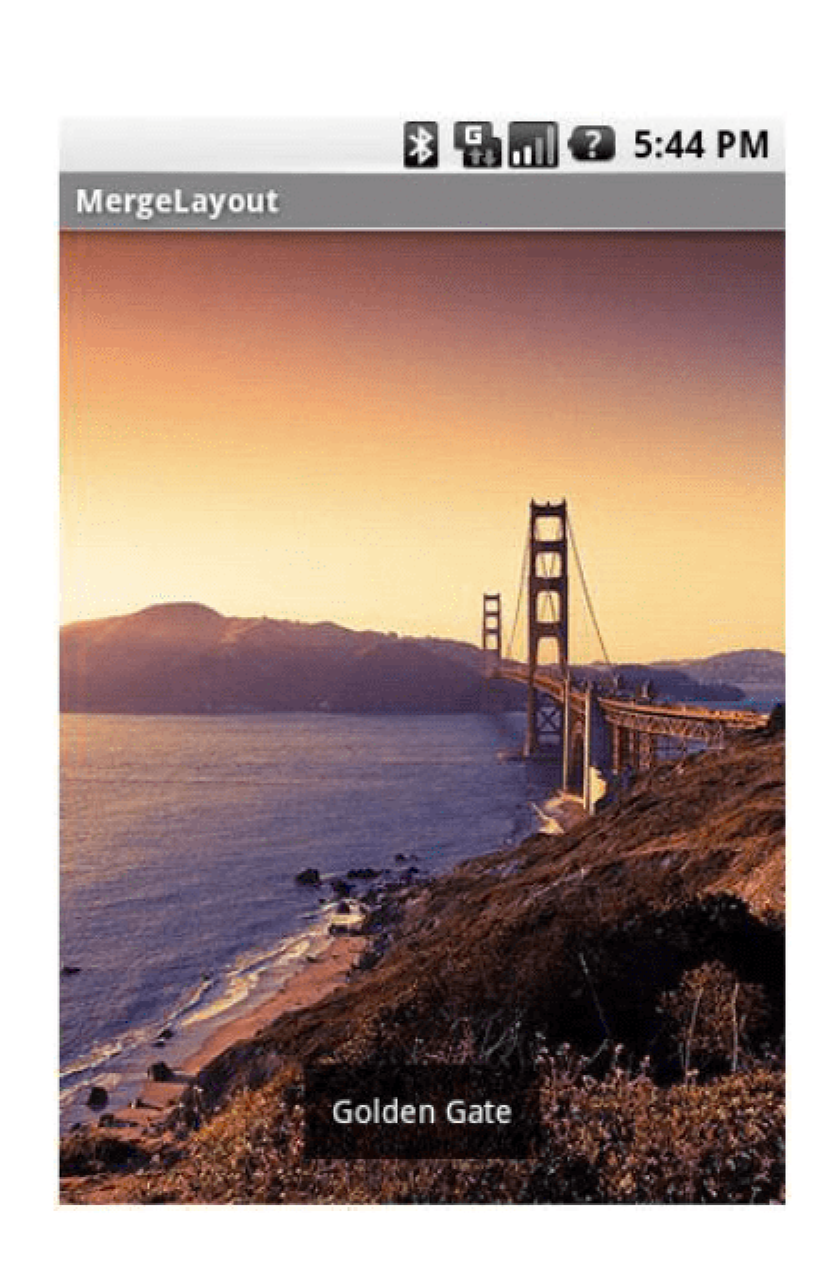


图 4-24　布局渲染效果

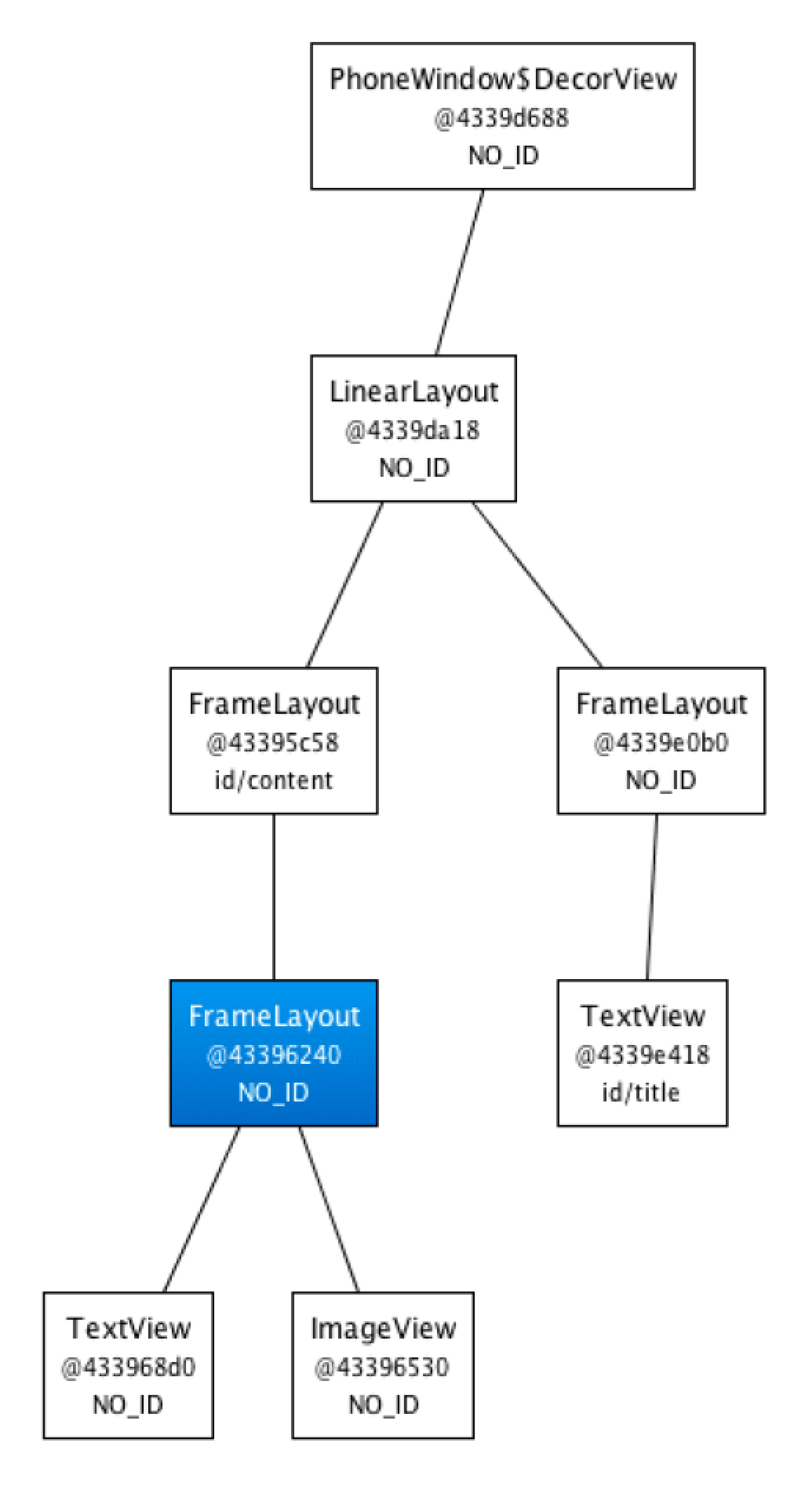


图 4-25　优化工具中的提示

在上述新代码中，TextView 和 ImageView 都直接添加到上一层的 FrameLayout 里。虽然视觉上看起来一样，但 View 的层次更加简单了。此时的 UI 结构视图如图 4-26 所示。

 UI 布局优化

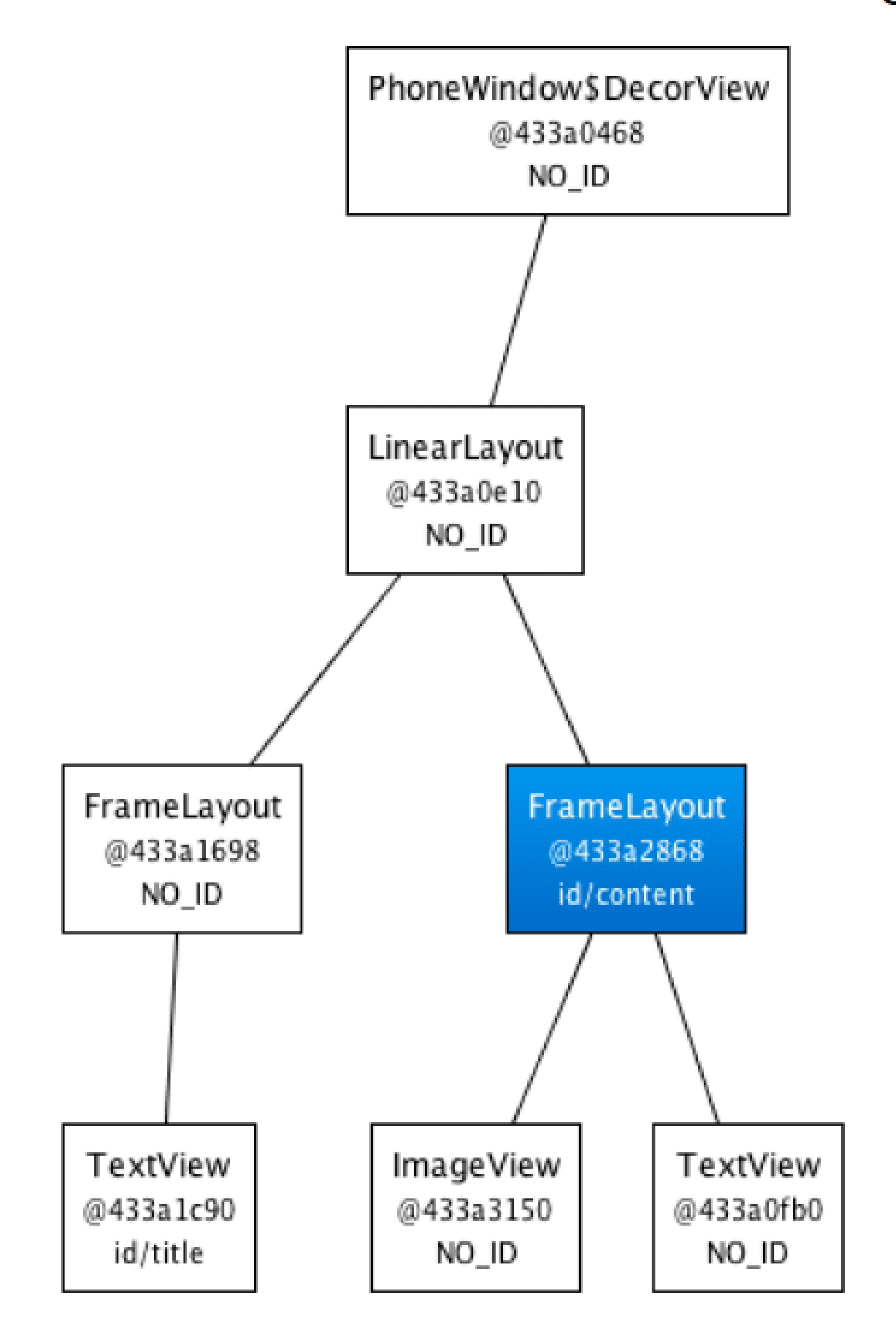


图 4-26　新的 UI 结构视图

很显然，在这个场合使用<merge/>标签是因为 Activity 的 ContentView 的父元素始终是 FrameLayout。如果我们的布局使用 LinearLayout 作为其根标签，那么就不能使用这个技巧。<merge/>标签在其他的一些场合也很有用。例如，它与<include/>标签结合起来就能表现得很完美。另外我们还可以在创建一个自定义的组合 View 时使用<merge/>。让我们看一个使用<merge/>创建一个新 View 的例子——OkCancelBar，它包含两个按钮，并可以设置按钮标签。下面的 XML 用于在一个图片上显示自定义的 View：

```
<merge
   xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:okCancelBar="http://schemas.android.com/apk/res/com.example.android.
merge">
   <ImageView
      android:layout width="fill parent"
      android:layout height="fill parent"
      android:scaleType="center"
      android:src="@drawable/golden gate" />
   <com.example.android.merge.OkCancelBar
      android:layout width="fill parent"
      android:layout height="wrap content"
      android:layout gravity="bottom"
      android:paddingTop="8dip"
      android:gravity="center horizontal"
      android:background="#AA000000"
      okCancelBar:okLabel="Save"
      okCancelBar:cancelLabel="Don't save" />
</merge>
```

新的布局效果如图 4-27 所示。

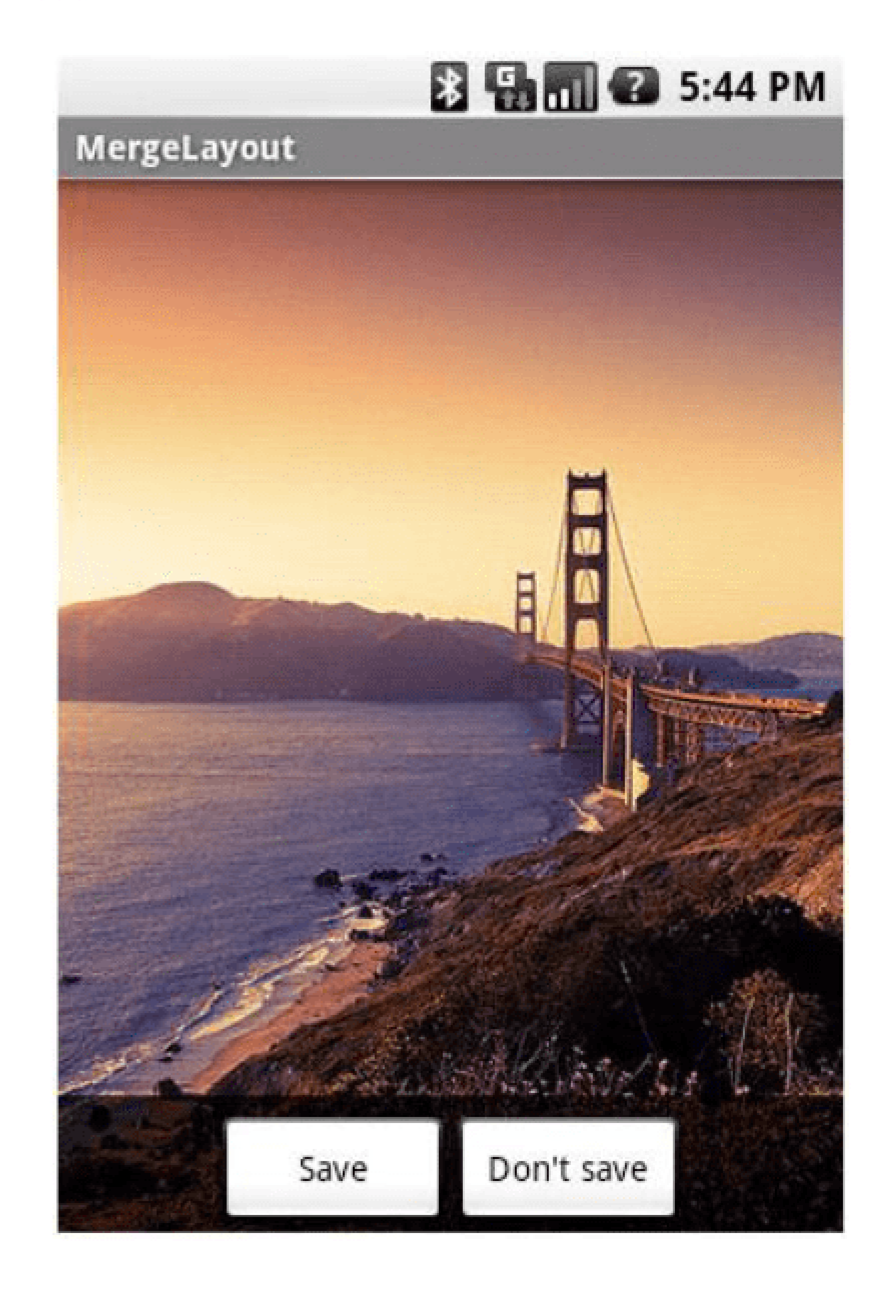


图 4-27　新的布局效果

OkCancelBar 部分的代码非常简单，因为这两个按钮在外部的 XML 文件中定义，通过类 LayoutInflate 导入。如下面的演示代码片段所示，R.layout.okcancelbar 以 OkCancelBar 作为父元素。

```
public class OkCancelBar extends LinearLayout {
   public OkCancelBar(Context context, AttributeSet attrs) {
      super(context, attrs);
      setOrientation(HORIZONTAL);
      setGravity(Gravity.CENTER);
      setWeightSum(1.0f);
      LayoutInflater.from(context).inflate(R.layout.okcancelbar, this, true);
      TypedArray array = context.obtainStyledAttributes(attrs,
R.styleable.OkCancelBar, 0, 0);
      String text = array.getString(R.styleable.OkCancelBar okLabel);
      if (text == null) text = "Ok";
       ((Button) findViewById(R.id.okcancelbar ok)).setText(text);
       text = array.getString(R.styleable.OkCancelBar_cancelLabel);
       if (text == null) text = "Cancel";
       ((Button) findViewById(R.id.okcancelbar cancel)).setText(text);
       array.recycle();
   }
}
```

而两个按钮的定义正如下面的 XML 代码所示，在此使用<merge/>标签直接添加两个按钮到 OkCancelBar。每个按钮都是从外部相同的 XML 布局文件包含进来的，这样做的好处是便于维护，我们只是简单地重写它们的 id。

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">
   <include
      layout="@layout/okcancelbar button"
      android:id="@+id/okcancelbar ok" />
   <include
      layout="@layout/okcancelbar button"
      android:id="@+id/okcancelbar cancel" />
</merge>
```

由此可见，我们创建了一个灵活且易于维护的自定义 View，它有着高效的 View 层次，如图 4-28 所示。

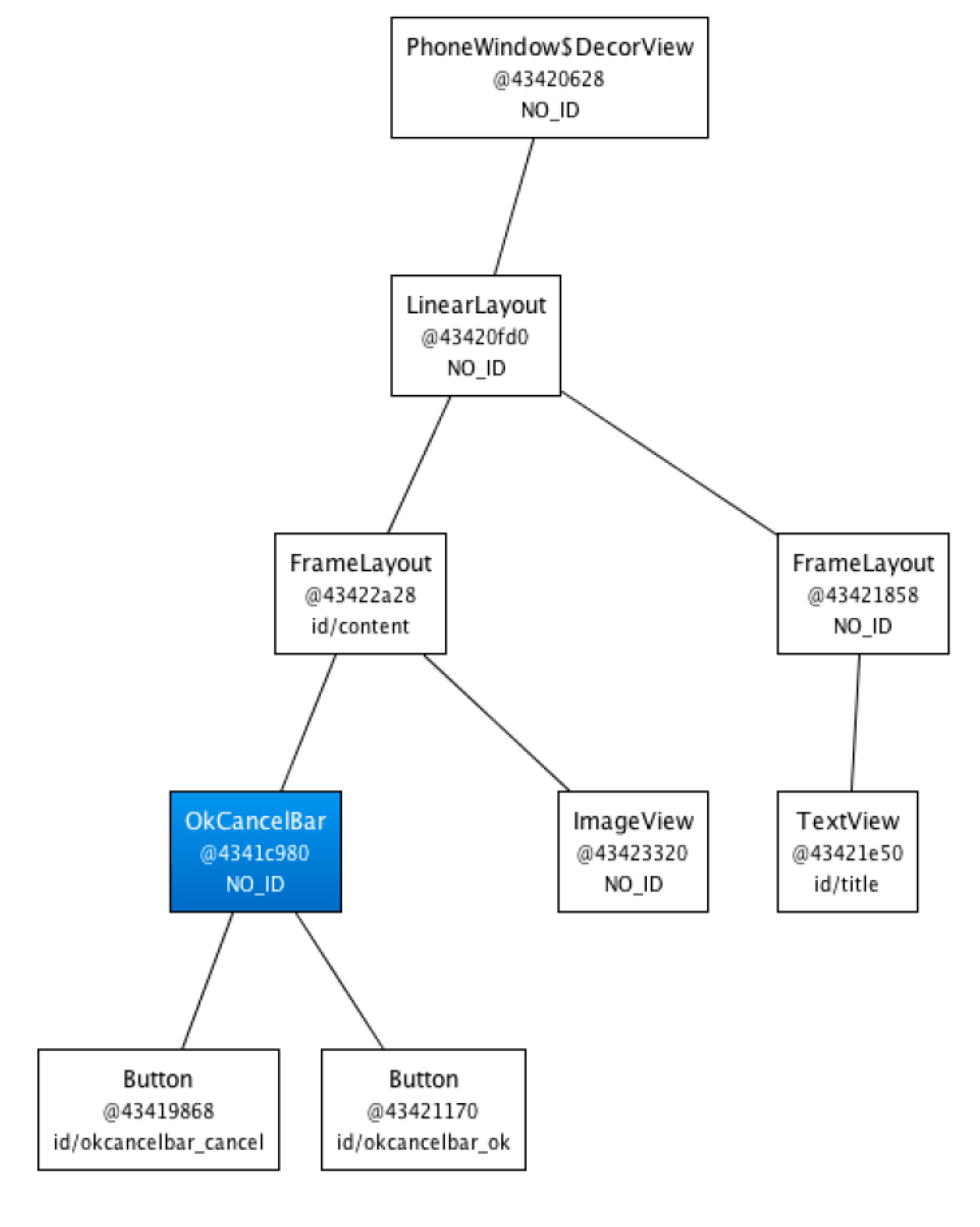


图 4-28 UI 结构视图

4.8 总结 Android UI 布局优化的原则和方法

根据本章前面内容的学习，Android UI 布局的基本知识已介绍完毕。本节将详细总结 Android UI 布局优化的原则和方法。

Android UI 布局优化主要依据下面的原则。

(1) 避免不必要的嵌套，不要把一个布局放置在其他布局里，除非是必要的。

(2) 避免使用太多视图，在一个布局中每增加一个新的视图，都会在过度操作时消耗很多资源。任何时候都不要在一个布局中包含超过 80 个视图，否则，消耗在过度操作的时间会很多。

(3) 避免深度嵌套，布局可以任意嵌套，这很容易创建复杂和深度嵌套的布局层次。如果没有硬件限制，将嵌套限制在 10 层以下是最好的实践。

从上面三点优化原则中可总结为：布局的优化主要是深度和广度，深度的表现主要在于布局的嵌套使用，广度的表现主要是包含过多的视图。

在开发过程中，单个 Activity 显示的自定义视图个数大多数会少于 20，层数少于 4 层。只有极少数的太大、太特殊的项目会超过 20 个视图，超过 4 层，这时就必须进行优化设计了。假如要设计一个拨号界面，其中包括 0～9、*、#等字符，是按 LinearLayout 来嵌套 Button 实现，还是用一个 GridView 来减少 Button 的个数从而减少冗余的代码？答案是建议使用 GridView 来实现。

Android

第5章

Android 的内存系统

我们知道，内存是计算机中重要的部件之一，它是与 CPU 进行沟通的桥梁。内存(Memory)也被称为内存储器，其作用是用于暂时存放 CPU 中的运算数据，以及与硬盘等外部存储器交换的数据。只要计算机在运行中，CPU 就会把需要运算的数据调到内存中进行运算，当运算完成后 CPU 再将结果传送出来。内存的运行也决定了计算机的稳定运行。其实智能手机就是一台微型的 PC，也具有和计算机一样的结构，例如 CPU 和内存。在本章的内容中，将和大家一起探讨 Android 内存系统的基本知识，为步入后面的内存优化知识做准备。

5.1 内存和进程的关系

手机内存的最直接作用体现在进程管理上，在本书前面的 2.2.3 节中已经讲解了 Android 程序生命周期的基本知识。在生命周期中，各种进程的启动、运行和结束都是通过内存实现的。读者需要明白，在 Android 中的进程和程序是两回事，程序可以一直保留在系统里，但是没有任何进程在后台“运行”，也不消耗任何系统资源。所有的程序保留在内存中，所有可以更快地启动回到它之前的状态。当你的内存用完了，系统会自动帮你杀掉你不用的任务。

另外读者还需要明白的是，Android 使用的是 RAM 方式，跟 Windows 的是两回事。在 Android 世界里，RAM 被用满了是一件“好”事，这意味着你可以快速打开之前打开的软件，回到之前的位置。所以 Android 很有效地使用 RAM，很多用户看到他们的 RAM 满了，就认为拖慢了他们的手机。而实际上，是你的 CPU——当你的软件真正运行时用到的空间——才是拖慢手机的瓶颈。

5.1.1 进程管理工具的纷争

当前市面中推出了很多进程管理工具，使用这些工具可以随时关闭正在运行的、不需要的进程，这样可以节约内存空间。但是如果依照“RAM 被用满了是一件好事”这一理论，会发现进程管理工具起了一个相反的作用。

其实这两者的争论自从 Android 诞生之日起就开始了，很流行的各种进程管理软件都声称帮我们释放内存是件好事，其实这是不正确的。当打开这些软件时，会告诉我们“运行”的软件和杀死他们的方法。但是我们也可以在“服务”里面看到到底程序的哪些部分在“运行”，占用了多少内存，剩余多少内存。所有的这一切都告诉我们，杀掉这些程序可以释放内存。但是这些软件都没有告诉我们这些程序到底消耗了多少 CPU 时钟，而仅仅告诉你能释放多少内存。实际上用满内存是一件好事，我们要注意的是 CPU，这才是真正影响消耗你的手机资源和电池的因素。

综上所述，杀掉程序通常是没有必要的，尤其是用“autokill”方式杀掉程序。更严重的是，这样做会更快的消耗掉我们的手机能力和电池性能。无论是手动杀掉进程，还是自动的杀掉进程，如果再次打开程序，实际上是在用 CPU 资源来做这件事。

事实上，这些进程管理软件不但消耗了系统资源，而且这些软件会莫名其妙地杀死其他程序，造成乱七八糟的结果。所有的这些告诉我们，我们的手机在用它自己的方式工作，用这些进程管理软件耽误的事情比得到的要多。

5.1.2 程序员的任务

既然进程管理工具的作用不大，我们程序员应该怎么办呢？在现实世界中，各种程序开发水平是不一样的。很多人以前或者现在使用这些进程管理软件释放内存，感觉手机快



 Android 的内存系统

了那么一点。造成这个问题的原因是，用的软件本身程序写得较差。比如，有得程序完全没有必要联网时还联着。这个时候，杀掉这些程序会得到好处。也就是说，只有你知道自己在干什么的时候，杀掉那些较差的程序才能有用。

事实上，很多开发者，包括 ROM 开发者，如果用了进程管理程序，当我们提交 bug 报告时，几乎看都不会看一眼，所以建议大家能不用就不要用了，除非真的知道自己在干什么。

如果真的关心自己手机的表现和进程，建议还是多关注下系统进程，看看里面说各种程序都消耗了多少资源，如果某个程序消耗太多，杀掉它可能会有一些帮助。

总体来说，进程管理软件正确的用途是杀那些出错的程序、会导致死机有 BUG 的进程以及疑似病毒进程等，而不是一味地追求内存的闲置空间多。

如果程序在内存里放着，而 CPU 不调用它们，这些程序就是死的。大多数在我们退出后就不再运行，不占用 CPU 资源(只有占用了 CPU 时间的才是要耗电的)，这就是 Android 2.2 以后版本系统中“快速启动”的工作原理。

5.1.3 Android 系统内存设计

Android 系统的特点是不需要太多的剩余内存，其实很多人都是把使用其他系统的习惯带过来了。其实 Android 的大多应用没有退出的设计其实是有原因的，这和系统对进程的调度机制有关系，这一点和 Java 机制是非常相似的。其实 Android 和 Java 的垃圾回收机制类似，在系统中有一个规则来回收内存，通过阀值来进行内存调度。只有低于这个阀值，系统才会按一个列表来关闭用户不需要的东西。

当然这个值默认设置得很小，所以会看到内存经常在很少的数值徘徊。但事实上他并不影响速度，而相反加快了下次启动应用的速度。这本来就是 Android 标榜的优势之一，如果人为去关闭进程，没有太大必要。特别是自动关进程的软件。

到此为止，肯定有人会反问道：为什么内存少的时候运行大型程序会慢呢？其实很简单，在内存剩余不多时打开大型程序，会触发系统自身的进程调度策略，这是十分消耗系统资源的操作，特别是在一个程序频繁向系统申请内存的时候。这种情况下系统并不会关闭所有打开的进程，而是选择性关闭，频繁地调度自然会拖慢系统。所以，论坛上有个更改内存阈值的程序可以有一定改善。但是这些改动可能会带来一些问题，这具体取决于值的设定。

再次回到 5.1.1 节中的问题，进程管理工具到底有无必要呢？不能说绝对有或没有，例如在运行大型程序之前，我们可以手动关闭一些进程释放内存，可以显著地提高运行速度，这是有用的。但是一些小程序，完全可交由系统自己管理，此时就是无用的。

谈到这里，可能有的读者会问，如果不关闭程序是不是会更耗电。在此说一下 Android 后台的原理就会明白。Android 的应用在被切换到后台时，它其实已经被暂停了，并不会消耗 CPU 资源，只是保留了运行状态。所以为什么有的程序切出去重进会到主界面。但是，一个程序如果想要在后台处理些东西，如音乐播放，它就会开启一个新服务。这个服务可以在后台持续运行，所以在后台耗电的也只有带服务的应用了。这个在进程管理软件里能看到，标签是 Service。所以没有带服务的应用在后台是完全不耗电的，没有必

要关闭。这种设计本来就是一个非常好的设计，当下次启动程序时会更快，因为不需要读取界面资源，所以说没有必要关掉它们，抹杀这个 Android 的优点。还有一个问题，为什么 Android 一个应用看起来那么耗内存。大家知道，Android 上的应用是 Java，当然需要虚拟机，而 Android 上的应用是带有独立虚拟机的，也就是每开一个应用就会打开一个独立的虚拟机。这样设计的原因是可以避免虚拟机崩溃导致整个系统崩溃，但代价就是需要更多内存。

以上这些设计确保了 Android 的稳定性，在正常情况下最多会崩溃单个程序，但是整个系统不会崩溃，也永远没有内存不足的提示出现。其实大家可能是受 Windows 系统的深远影响，总想保留更多的内存，但实际上这并不一定会提升速度，相反却丧失了程序启动快的这一系统特色，这很没有必要。大家不妨按我说的习惯来使用 Android 系统。

5.2 分析 Android 的进程通信机制

要想实现对 Android 系统内存的优化，需要首先了解 Android 的内存系统，了解内存控制进程运行的机制。在本节的内容中，将带领大家一起探讨分析 Android 的进程通信机制。

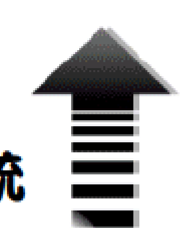
5.2.1 Android 的进程间通信(IPC)机制 Binder

在 Android 系统中，每一个应用程序都是由一些 Activity 和 Service 组成的，一般 Service 运行在独立的进程中，而 Activity 有可能运行在同一个进程中，也有可能运行在不同的进程中。那么不在同一个进程的 Activity 或者 Service 之间究竟是如何通信的呢？下面将介绍的 Binder 进程间通信机制来实现这个功能。

众所周知，Android 系统是基于 Linux 内核的，而 Linux 内核继承和兼容了丰富的 Unix 系统进程间通信(IPC)机制。有传统的管道(Pipe)、信号(Signal)和跟踪(Trace)，这三项通信手段只能用于父进程和子进程之间，或者只用于兄弟进程之间。随着技术的发展，后来又增加了命令管道(Named Pipe)，这样使得进程之间的通信不再局限于父子进程或者兄弟进程之间。为了更好地支持商业应用中的事务处理，在 AT&T 的 Unix 系统 V 中，又增加了如下三种称为“System V IPC”的进程间通信机制：

- 报文队列(Message)
- 共享内存(Share Memory)
- 信号量(Semaphore)

后来 BSD Unix 对“System V IPC”机制进行了重要的扩充，提供了一种称为插口(Socket)的进程间通信机制。但是 Android 系统没有采用上述提到的各种进程间通信机制，而是采用 Binder 机制，这难道仅仅是因为考虑到了移动设备硬件性能较差、内存较低的特点吗？这只有 Android 工程师知道，我们不得而知。Binder 其实也不是 Android 提出来的一套新的进程间通信机制，它是基于 OpenBinder 来实现的。OpenBinder 最先是由 Be Inc. 开发的，后来 Palm Inc.也跟着借鉴使用。现在 OpenBinder 的作者 Dianne Hackborn 就是在



 Android 的内存系统

Google 工作，负责 Android 平台的开发工作。

再次强调一下，Binder 是一种进程间通信机制，这是一种类似于 COM 和 CORBA 分布式组件架构。通俗点说，其实是提供远程过程调用(RPC)功能。从英文字面上意思看，Binder 具有黏结剂的意思，那么它把什么东西黏结在一起呢？在 Android 系统的 Binder 机制中，由一系统组件组成，分别是 Client、Server、Service Manager 和 Binder 驱动程序，其中 Client、Server 和 Service Manager 运行在用户空间，Binder 驱动程序运行内核空间。Binder 就是一种把这 4 个组件黏合在一起的黏结剂了，其中，核心组件便是 Binder 驱动程序了，Service Manager 提供了辅助管理的功能，Client 和 Server 正是在 Binder 驱动和 Service Manager 提供的基础设施上，进行 Client/Server 之间的通信。Service Manager 和 Binder 驱动已经在 Android 平台中实现完毕，开发者只要按照规范实现自己的 Client 和 Server 组件即可。但是说起来简单，具体做起来却很难。对初学者来说，Android 系统的 Binder 机制是最难理解的了，而 Binder 机制无论从系统开发还是应用开发的角度来看，都是 Android 系统中最重要的组成，所以很有必要深入了解 Binder 的工作方式。要深入了解 Binder 的工作方式，最好的方式是阅读 Binder 相关的源代码了，Linux 的鼻祖 Linus Torvalds 曾经说过一句名言“RTFSC：Read The Fucking Source Code”。

虽说阅读 Binder 的源代码是学习 Binder 机制的最好方式，但是也绝不能打无准备之仗，因为 Binder 的相关源代码是比较枯燥无味而且难以理解的，如果能够辅予一些理论知识那就更好了。

要想理解 Binder 机制，必须了解 Binder 在用户空间的三个组件 Client、Server 和 Service Manager 之间的相互关系，了解内核空间中 Binder 驱动程序的数据结构和设计原理。非常感谢这两位作者给我们带来这么好的 Binder 学习资料。具体来说，Android 系统 Binder 机制中的 4 个组件 Client、Server、Service Manager 和 Binder 驱动程序的关系如图 5-1 所示。

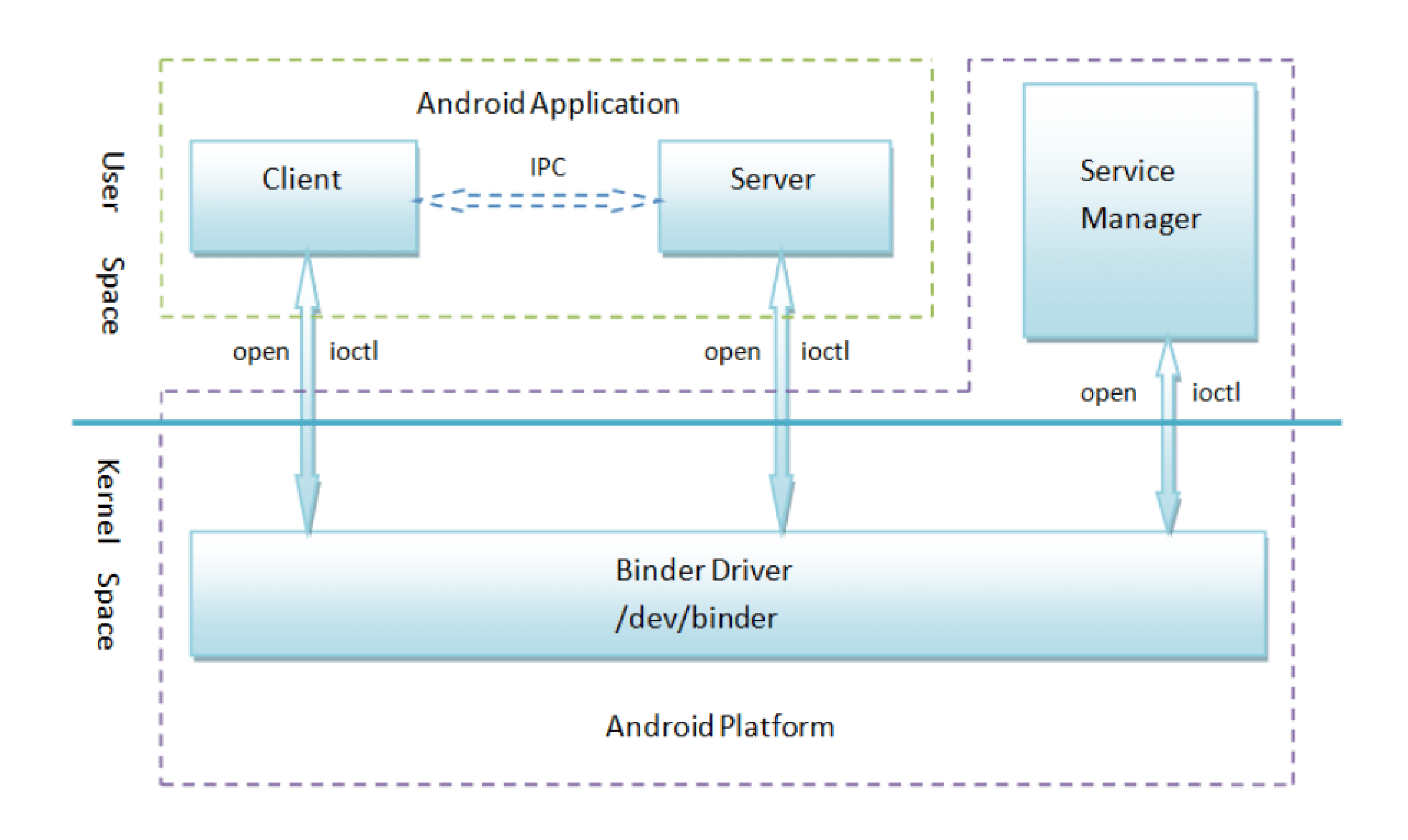


图 5-1　组件 Client、Server、Service Manager 和 Binder 驱动程序的关系

对图 5-1 所示关系的具体说明如下。

(1) Client、Server 和 Service Manager 实现在用户空间中，Binder 驱动程序实现在内核空间中。

(2) Binder 驱动程序和 Service Manager 在 Android 平台中已经实现，开发者只需要在用户空间实现自己的 Client 和 Server。

(3) Binder 驱动程序提供设备文件“/dev/binder”与用户空间交互，Client、Server 和 Service Manager 通过文件操作函数 open()和 ioctl()与 Binder 驱动程序进行通信。

(4) Client 和 Server 之间的进程间通信通过 Binder 驱动程序间接实现。

(5) Service Manager 是一个守护进程，用来管理 Server，并向 Client 提供查询 Server 接口的能力。

5.2.2 Service Manager 是 Binder 机制的上下文管理者

在分析 Binder 源代码时，需要先弄清楚 Service Manager 是如何告知 Binder 驱动程序它是 Binder 机制的上下文管理者。Service Manager 是整个 Binder 机制的守护进程，用来管理开发者创建的各种 Server，并且向 Client 提供查询 Server 远程接口的功能。

因为 Service Manager 组件是用来管理 Server 并且向 Client 提供查询 Server 远程接口的功能，所以 Service Manager 必然要和 Server 以及 Client 进行通信。我们知道，Service Manger、Client 和 Server 三者分别是运行在独立的进程当中的，这样它们之间的通信也属于进程间的通信，而且也是采用 Binder 机制进行进程间通信。因此，Service Manager 在充当 Binder 机制的守护进程的角色的同时，也在充当 Server 的角色，但是它是一种特殊的 Server，要想了解具体的特殊之处请看下面的内容。

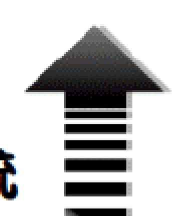
与 Service Manager 相关的源代码较多，本书不会完整地去分析每一行代码，主要是带着 Service Manager 是如何成为整个 Binder 机制中的守护进程这条主线来分析相关源代码，这主要包括从用户空间到内核空间的相关源代码。

Service Manager 在用户空间的源代码位于 frameworks/base/cmds/servicemanager 目录下，主要是由文件 binder.h、binder.c 和 service_manager.c 组成。Service Manager 的入口位于文件 service_manager.c 中的函数 main()中，代码如下。

```
int main(int argc, char **argv){
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;
    bs = binder_open(128*1024);
    if (binder_become_context_manager(bs)) {
        LOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }
    svcmgr_handle = svcmgr;
    binder_loop(bs, svcmgr_handler);
    return 0;
}
```

上述函数 main()主要有如下三个功能：

 Android 的内存系统



- 打开 Binder 设备文件；
- 告诉 Binder 驱动程序自己是 Binder 上下文管理者，即我们前面所说的守护进程；
- 进入一个无穷循环，充当 Server 的角色，等待 Client 的请求。

在进入上述三个功能之间，先来看一下这里用到的结构体 binder_state、宏 BINDER_SERVICE_MANAGER 的定义。结构体 binder_state 定义在文件 frameworks/base/cmds/servicemanager/binder.c 中，代码如下：

```
struct binder_state {
    int fd;
    void *mapped;
    unsigned mapsize;
};
```

其中 fd 表示文件描述符，即表示打开的“/dev/binder”设备文件描述符；mapped 表示把设备文件“/dev/binder”映射到进程空间的起始地址；mapsize 表示上述内存映射空间的大小。

宏 BINDER_SERVICE_MANAGER 在文件 frameworks/base/cmds/servicemanager/binder.h 中定义，代码如下：

```
/* the one magic object */
#define BINDER_SERVICE_MANAGER ((void*) 0)
```

这表示 Service Manager 的句柄为 0。Binder 通信机制使用句柄来代表远程接口，此句柄的意义和 Windows 编程中用到的句柄是差不多。前面说到，Service Manager 在充当守护进程的同时，它充当 Server 的角色，当它作为远程接口使用时，它的句柄值便为 0，这就是它的特殊之处，其余的 Server 的远程接口句柄值都是一个大于 0 而且由 Binder 驱动程序自动进行分配的。

函数首先打开 Binder 设备文件的操作函数 binder_open()，此函数的定义位于文件 frameworks/base/cmds/servicemanager/binder.c 中，代码如下：

```
struct binder_state *binder_open(unsigned mapsize){
    struct binder_state *bs;
    bs = malloc(sizeof(*bs));
    if (!bs) {
        errno = ENOMEM;
        return 0;
    }
    bs->fd = open("/dev/binder", O_RDWR);
    if (bs->fd < 0) {
        fprintf(stderr,"binder: cannot open device (%s)\n",
                strerror(errno));
        goto fail_open;
    }
    bs->mapsize = mapsize;
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
    if (bs->mapped == MAP_FAILED) {
        fprintf(stderr,"binder: cannot map device (%s)\n",
                strerror(errno));
```

```
        goto fail map;
    }
        /* TODO: check version */
    return bs;
fail map:
    close(bs->fd);
fail open:
    free(bs);
    return 0;
}
```

通过文件操作函数 open()打开设备文件“/dev/binder”，此设备文件是在 Binder 驱动程序模块初始化的时候创建的。接下来先看一下这个设备文件的创建过程，来到 kernel/common/drivers/staging/android 目录，打开文件 binder.c，可以看到如下模块初始化入口 binder_init：

```
static struct file operations binder fops = {
    .owner = THIS MODULE,
    .poll = binder poll,
    .unlocked ioctl = binder ioctl,
    .mmap = binder mmap,
    .open = binder open,
    .flush = binder flush,
    .release = binder release,
};

static struct miscdevice binder miscdev = {
    .minor = MISC DYNAMIC MINOR,
    .name = "binder",
    .fops = &binder fops
};

static int   init binder init(void)
{
    int ret;

    binder proc dir entry root = proc mkdir("binder", NULL);
    if (binder proc dir entry root)
        binder proc dir entry proc = proc mkdir("proc",
binder proc dir entry root);
    ret = misc register(&binder miscdev);
    if (binder proc dir entry root) {
        create proc read entry("state", S IRUGO,
binder proc dir entry root, binder read proc state, NULL);
        create proc read entry("stats", S IRUGO,
binder proc dir entry root, binder read proc stats, NULL);
        create proc read entry("transactions", S IRUGO,
binder proc dir entry root, binder read proc transactions, NULL);
        create_proc_read_entry("transaction_log", S_IRUGO,
binder proc dir entry root, binder read proc transaction log,
&binder_transaction_log);
```

 Android 的内存系统

```
        create proc read entry("failed transaction log", S IRUGO,
binder proc dir entry root, binder read proc transaction log,
&binder transaction log failed);
    }
    return ret;
}
device_initcall(binder_init);
```

在函数 misc_register()中实现了创建设备文件的功能，并实现了 misc 设备的注册工作，在/proc 目录中创建了各种 Binder 相关的文件供用户访问。从设备文件的操作方法 binder_fops 可以看出，通过如下函数 binder_open 的执行语句：

```
bs->fd = open("/dev/binder", O_RDWR);
```

即可进入到 Binder 驱动程序的 binder_open()函数：

```
static int binder open(struct inode *nodp, struct file *filp)
{
    struct binder proc *proc;

    if (binder debug mask & BINDER DEBUG OPEN CLOSE)
        printk(KERN INFO "binder open: %d:%d\n", current->group leader-
>pid, current->pid);

    proc = kzalloc(sizeof(*proc), GFP KERNEL);
    if (proc == NULL)
        return -ENOMEM;
    get task struct(current);
    proc->tsk = current;
    INIT LIST HEAD(&proc->todo);
    init waitqueue head(&proc->wait);
    proc->default priority = task nice(current);
    mutex lock(&binder lock);
    binder stats.obj created[BINDER STAT PROC]++;
    hlist add head(&proc->proc node, &binder procs);
    proc->pid = current->group leader->pid;
    INIT LIST HEAD(&proc->delivered death);
    filp->private data = proc;
    mutex unlock(&binder lock);

    if (binder proc dir entry proc) {
        char strbuf[11];
        snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
        remove proc entry(strbuf, binder proc dir entry proc);
        create proc read entry(strbuf, S IRUGO,
binder proc dir entry proc, binder read proc proc, proc);
    }
    return 0;
}
```

上述函数的主要作用是创建一个名为 binder_proc 的数据结构，用此数据结构来保存打

开设备文件“/dev/binder”的进程的上下文信息，并且将这个进程上下文信息保存在打开文件结构 file 的私有数据成员变量 private_data 中。这样当在执行其他文件操作时，就通过打开文件结构 file 来取回这个进程上下文信息了。这个进程上下文信息同时还会保存在一个全局哈希表 binder_procs 中，供驱动程序内部使用。哈希表 binder_procs 定义在文件的开头：

```
static HLIST_HEAD(binder_procs);
```

而结构体 struct binder_proc 也被定义在文件 kernel/common/drivers/staging/android/binder.c 中：

```
struct binder_proc {
    struct hlist node proc node;
    struct rb root threads;
    struct rb root nodes;
    struct rb root refs by desc;
    struct rb root refs by node;
    int pid;
    struct vm area struct *vma;
    struct task struct *tsk;
    struct files struct *files;
    struct hlist node deferred work node;
    int deferred_work;
    void *buffer;
    ptrdiff t user buffer offset;
    struct list head buffers;
    struct rb root free buffers;
    struct rb root allocated buffers;
    size t free async space;
    struct page **pages;
    size t buffer size;
    uint32 t buffer free;
    struct list head todo;
    wait queue head t wait;
    struct binder stats stats;
    struct list head delivered death;
    int max threads;
    int requested threads;
    int requested threads started;
    int ready threads;
    long default priority;
};
```

上述结构体的成员比较多，其中最终重要的有 4 个成员变量：

- threads
- nodes
- refs_by_desc
- refs_by_node

上述 4 个成员变量都是表示红黑树的节点，也就是说，binder_proc 分别挂在 4 个红黑

 Android 的内存系统



树下，具体说明如下。

- threads 树：用来保存 binder_proc 进程内用于处理用户请求的线程，它的最大数量由 max_threads 来决定；
- node 树：用来保存 binder_proc 进程内的 Binder 实体；
- refs_by_desc 树和 refs_by_node 树：用来保存 binder_proc 进程内的 Binder 引用，即引用其他进程的 Binder 实体，它分别用两种方式来组织红黑树：一种是以句柄作为 key 值来组织；另一种是以引用的实体节点的地址值作为 key 值来组织。它们都是表示同一样东西，只不过是为了内部查找方便而用两个红黑树来表示。

这样，打开设备文件/dev/binder 的操作就完成了，接下来需要对打开的设备文件进行内存映射操作 mmap。

```
bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
```

对应 Binder 驱动程序的是函数 binder_mmap()，实现代码如下：

```
static int binder mmap(struct file *filp, struct vm area struct *vma)
{
    int ret;
    struct vm struct *area;
    struct binder proc *proc = filp->private data;
    const char *failure string;
    struct binder buffer *buffer;
    if ((vma->vm end - vma->vm start) > SZ 4M)
        vma->vm end = vma->vm start + SZ 4M;
    if (binder debug mask & BINDER DEBUG OPEN CLOSE)
        printk(KERN INFO
            "binder_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
            proc->pid, vma->vm start, vma->vm end,
            (vma->vm end - vma->vm start) / SZ 1K, vma->vm flags,
            (unsigned long)pgprot val(vma->vm page prot));
    if (vma->vm flags & FORBIDDEN MMAP FLAGS) {
        ret = -EPERM;
        failure string = "bad vm flags";
        goto err bad arg;
    }
    vma->vm flags = (vma->vm flags | VM DONTCOPY) & ~VM MAYWRITE;

    if (proc->buffer) {
        ret = -EBUSY;
        failure string = "already mapped";
        goto err already mapped;
    }

    area = get vm area(vma->vm end - vma->vm start, VM IOREMAP);
    if (area == NULL) {
        ret = -ENOMEM;
        failure string = "get vm area";
        goto err get vm area failed;
    }
```

```
    proc->buffer = area->addr;
    proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;

#ifdef CONFIG_CPU_CACHE_VIPT
    if (cache_is_vipt_aliasing()) {
        while (CACHE_COLOUR((vma->vm_start ^ (uint32_t)proc->buffer))) {
            printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p bad
alignment\n", proc->pid, vma->vm_start, vma->vm_end, proc->buffer);
            vma->vm_start += PAGE_SIZE;
        }
    }
#endif
    proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma-
>vm_start) / PAGE_SIZE), GFP_KERNEL);
    if (proc->pages == NULL) {
        ret = -ENOMEM;
        failure_string = "alloc page array";
        goto err_alloc_pages_failed;
    }
    proc->buffer_size = vma->vm_end - vma->vm_start;

    vma->vm_ops = &binder_vm_ops;
    vma->vm_private_data = proc;

    if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer +
PAGE_SIZE, vma)) {
        ret = -ENOMEM;
        failure_string = "alloc small buf";
        goto err_alloc_small_buf_failed;
    }
    buffer = proc->buffer;
    INIT_LIST_HEAD(&proc->buffers);
    list_add(&buffer->entry, &proc->buffers);
    buffer->free = 1;
    binder_insert_free_buffer(proc, buffer);
    proc->free_async_space = proc->buffer_size / 2;
    barrier();
    proc->files = get_files_struct(current);
    proc->vma = vma;

    /*printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p\n", proc->pid,
vma->vm_start, vma->vm_end, proc->buffer);*/
    return 0;

err_alloc_small_buf_failed:
    kfree(proc->pages);
    proc->pages = NULL;
err_alloc_pages_failed:
    vfree(proc->buffer);
    proc->buffer = NULL;
err_get_vm_area_failed:
```

 Android 的内存系统



```
err already mapped:
err bad arg:
    printk(KERN ERR "binder mmap: %d %lx-%lx %s failed %d\n", proc->pid,
vma->vm start, vma->vm end, failure string, ret);
    return ret;
}
```

在上述函数 binder_mmap()中，首先通过 filp->private_data 得到在打开设备文件“/dev/binder”时创建的结构 binder_proc。内存映射信息放在 vma 参数中。读者需要注意，这里的 vma 的数据类型是结构 vm_area_struct，它表示的是一块连续的虚拟地址空间区域。在函数变量声明的地方，我们还看到有一个类似的结构体 vm_struct，这个数据结构也是表示一块连续的虚拟地址空间区域。那么，这两者的区别是什么呢？在 Linux 系统中，结构体 vm_area_struct 表示的虚拟地址是给进程使用的，而结构体 vm_struct 表示的虚拟地址是给内核使用的，它们对应的物理页面都可以是不连续的。结构体 vm_area_struct 表示的地址空间范围是 0～3G，而结构体 vm_struct 表示的地址空间范围是(3G + 896M + 8M)～4G。为什么结构体 vm_struct 表示的地址空间范围不是 3～4G 呢？因为 3G～(3G + 896M)范围的地址是用来映射连续的物理页面的，这个范围的虚拟地址和对应的实际物理地址有着简单的对应关系，即对应 0～896M 的物理地址空间，而(3G + 896M)～(3G + 896M + 8M)是安全保护区域。例如所有指向这 8M 地址空间的指针都是非法的，所以结构体 vm_struct 使用(3G + 896M + 8M)～4G 地址空间来映射非连续的物理页面。

此处为什么会同时使用进程虚拟地址空间和内核虚拟地址空间来映射同一个物理页面呢？这就是 Binder 进程间通信机制的精髓所在了。在同一个物理页面，一方映射到进程虚拟地址空间，一方面映射到内核虚拟地址空间，这样进程和内核之间就可以减少一次内存拷贝工作，提高了进程之间的通信效率。

讲解了 binder_mmap 的原理之后，整个函数的逻辑就很好理解了。但是在此还是先要解释一下 binder_proc 结构体中的如下成员变量。

- buffer：是一个 void*指针，它表示要映射的物理内存在内核空间中的起始位置。
- buffer_size：是一个 size_t 类型的变量，表示要映射的内存的大小。
- pages：是一个 struct page*类型的数组，struct page 是用来描述物理页面的数据结构。
- user_buffer_offset：是一个 ptrdiff_t 类型的变量，它表示的是内核使用的虚拟地址与进程使用的虚拟地址之间的差值，即如果某个物理页面在内核空间中对应的虚拟地址是 addr 的话，那么这个物理页面在进程空间对应的虚拟地址就为“addr + user_buffer_offset”格式。

接下来还需要看一下 Binder 驱动程序管理内存映射地址空间的方法，即如何管理 buffer ~ (buffer + buffer_size)这段地址空间的，这个地址空间被划分为一段一段来管理，每一段是用结构体 binder_buffer 来描述的，代码如下：

```
struct binder buffer {
    struct list head entry; /* free and allocated entries by addesss */
    struct rb node rb node; /* free entry by size or allocated entry */
                /* by address */
    unsigned free : 1;
```

```
    unsigned allow user free : 1;
    unsigned async transaction : 1;
    unsigned debug id : 29;
    struct binder transaction *transaction;
    struct binder node *target node;
    size t data size;
    size t offsets size;
    uint8 t data[0];
};
```

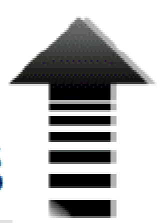
每一个 binder_buffer 通过其成员 entry 按从低地址到高地址连入到 struct binder_proc 中的 buffers 表示的链表中，同时，每一个 binder_buffer 又分为正在使用的和空闲的，通过 free 成员变量来区分，空闲的 binder_buffer 通过成员变量 rb_node 的帮助，连入到 struct binder_proc 中的 free_buffers 表示的红黑树中。而那些正在使用的 binder_buffer，通过成员变量 rb_node 连入到 binder_proc 中的 allocated_buffers 表示的红黑树中。这样做当然是为了方便查询和维护这块地址空间了。

然后回到函数 binder_mmap()，首先是对参数做一些检查，例如要映射的内存大小不能超过 SIZE_4M，即 4M。在来到文件 service_manager.c 中的 main()函数，这里传进来的值是 128*1024 个字节，即 128K，这个检查没有问题。通过检查之后，调用函数 get_vm_area()获得一个空闲的 vm_struct 区间，并初始化 proc 结构体的 buffer、user_buffer_offset、pages 和 buffer_size 等成员变量，接着调用 binder_update_page_range 为虚拟地址空间 proc->buffer ~ proc->buffer + PAGE_SIZE 分配一个空闲的物理页面，同时这段地址空间使用一个 binder_buffer 来描述，分别插入到 proc->buffers 链表和 proc->free_buffers 红黑树中去，最后还初始化了 proc 结构体的 free_async_space、files 和 vma 三个成员变量。

然后继续分析函数 binder_update_page_range()，看一下 Binder 驱动程序是如何实现把一个物理页面同时映射到内核空间和进程空间去的。

```
static int binder update page range(struct binder proc *proc, int
allocate,
    void *start, void *end, struct vm area struct *vma)
{
    void *page addr;
    unsigned long user page addr;
    struct vm struct tmp area;
    struct page **page;
    struct mm struct *mm;
    if (binder debug mask & BINDER DEBUG BUFFER ALLOC)
        printk(KERN INFO "binder: %d: %s pages %p-%p\n",
              proc->pid, allocate ? "allocate" : "free", start, end);
    if (end <= start)
        return 0;
    if (vma)
        mm = NULL;
    else
        mm = get task mm(proc->tsk);
    if (mm) {
```

 Android 的内存系统



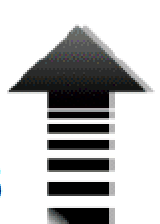
```
        down write(&mm->mmap sem);
        vma = proc->vma;
    }
    if (allocate == 0)
        goto free range;
    if (vma == NULL) {
        printk(KERN ERR "binder: %d: binder alloc buf failed to "
              "map pages in userspace, no vma\n", proc->pid);
        goto err no vma;
    }
    for (page addr = start; page addr < end; page addr += PAGE SIZE) {
        int ret;
        struct page **page array ptr;
        page = &proc->pages[(page addr - proc->buffer) / PAGE SIZE];
        BUG ON(*page);
        *page = alloc page(GFP KERNEL |   GFP ZERO);
        if (*page == NULL) {
            printk(KERN ERR "binder: %d: binder alloc buf failed "
                  "for page at %p\n", proc->pid, page addr);
            goto err alloc page failed;
        }
        tmp area.addr = page addr;
        tmp area.size = PAGE SIZE + PAGE SIZE /* guard page? */;
        page array ptr = page;
        ret = map vm area(&tmp area, PAGE KERNEL, &page array ptr);
        if (ret) {
            printk(KERN ERR "binder: %d: binder alloc buf failed "
                  "to map page at %p in kernel\n",
                  proc->pid, page addr);
            goto err map kernel failed;
        }
        user page addr =
            (uintptr t)page addr + proc->user buffer offset;
        ret = vm insert page(vma, user page addr, page[0]);
        if (ret) {
            printk(KERN ERR "binder: %d: binder alloc buf failed "
                  "to map page at %lx in userspace\n",
                  proc->pid, user page addr);
            goto err vm insert page failed;
        }
        /* vm insert page does not seem to increment the refcount */
    }
    if (mm) {
        up write(&mm->mmap sem);
        mmput(mm);
    }
    return 0;
free range:
    for (page addr = end - PAGE SIZE; page addr >= start;
         page addr -= PAGE SIZE) {
        page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
```

```
        if (vma)
            zap page range(vma, (uintptr t)page addr +
                proc->user buffer offset, PAGE SIZE, NULL);
err vm insert page failed:
        unmap kernel range((unsigned long)page addr, PAGE SIZE);
err map kernel failed:
          free page(*page);
        *page = NULL;
err alloc page failed:
        ;
    }
err no vma:
    if (mm) {
        up write(&mm->mmap sem);
        mmput(mm);
    }
    return -ENOMEM;
}
```

通过上述函数不但可以分配物理页面，而且可以用来释放物理页面，这可以通过参数 allocate 来区别，在此我们只需关注分配物理页面的情况。要分配物理页面的虚拟地址空间范围为(start～end)，函数前面的一些检查逻辑就不看了，我们只需直接看中间的 for 循环：

```
    for (page addr = start; page addr < end; page addr += PAGE SIZE) {
        int ret;
        struct page **page array ptr;
        page = &proc->pages[(page addr - proc->buffer) / PAGE SIZE];
        BUG ON(*page);
        *page = alloc page(GFP KERNEL |   GFP ZERO);
        if (*page == NULL) {
            printk(KERN ERR "binder: %d: binder alloc buf failed "
                  "for page at %p\n", proc->pid, page addr);
            goto err alloc page failed;
        }
        tmp area.addr = page addr;
        tmp area.size = PAGE SIZE + PAGE SIZE /* guard page? */;
        page array ptr = page;
        ret = map vm area(&tmp area, PAGE KERNEL, &page array ptr);
        if (ret) {
            printk(KERN ERR "binder: %d: binder alloc buf failed "
                  "to map page at %p in kernel\n",
                  proc->pid, page addr);
            goto err map kernel failed;
        }
        user page addr =
            (uintptr t)page addr + proc->user buffer offset;
        ret = vm insert page(vma, user page addr, page[0]);
        if (ret) {
            printk(KERN ERR "binder: %d: binder alloc buf failed "
                  "to map page at %lx in userspace\n",
```

 Android 的内存系统



```
                  proc->pid, user page addr);
            goto err vm insert page failed;
        }
        /* vm insert page does not seem to increment the refcount */
    }
```

在上述代码中，首先调用 alloc_page()分配一个物理页面，此函数返回一个结构体 page 物理页面描述符，根据这个描述的内容初始化好结构体 vm_struct tmp_area，然后通过 map_vm_area 将这个物理页面插入到 tmp_area 描述的内核空间去，接着通过 page_addr + proc->user_buffer_offset 获得进程虚拟空间地址，并通过函数 vm_insert_page()将这个物理页面插入到进程地址空间去，参数 vma 表示要插入的进程的地址空间。

这样，文件 frameworks/base/cmds/servicemanager/binder.c 中的函数 binder_open()讲解完毕。我们再次回到文件 frameworks/base/cmds/servicemanager/service_manager.c 中的 main()函数，接下来需要调用 binder_become_context_manager 来通知 Binder 驱动程序自己是 Binder 机制的上下文管理者，即守护进程。函数 binder_become_context_manager()位于文件 frameworks/base/cmds/servicemanager/binder.c 中，具体代码如下：

```
int binder become context manager(struct binder state *bs){
   return ioctl(bs->fd, BINDER SET CONTEXT MGR, 0);
}
```

在此通过调用 ioctl 文件操作函数通知 Binder 驱动程序自己是守护进程，命令号是 BINDER_SET_CONTEXT_MGR，并没有任何参数。BINDER_SET_CONTEXT_MGR 定义为：

```
#define BINDER_SET_CONTEXT_MGR      _IOW('b', 7, int)
```

这样就进入到 Binder 驱动程序的函数 binder_ioctl()，在此只关注如下 BINDER_SET_CONTEXT_MGR 命令即可：

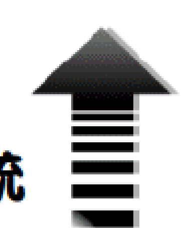
```
static long binder ioctl(struct file *filp, unsigned int cmd, unsigned
long arg)
{
    int ret;
    struct binder proc *proc = filp->private data;
    struct binder thread *thread;
    unsigned int size =  IOC SIZE(cmd);
    void   user *ubuf = (void   user *)arg;
    /*printk(KERN INFO "binder ioctl: %d:%d %x %lx\n", proc->pid,
current->pid, cmd, arg);*/
    ret = wait event interruptible(binder user error wait,
binder stop on user error < 2);
    if (ret)
        return ret;
    mutex lock(&binder lock);
    thread = binder get thread(proc);
    if (thread == NULL) {
        ret = -ENOMEM;
        goto err;
```

```
    }
    switch (cmd) {
       ......
    case BINDER SET CONTEXT MGR:
        if (binder context mgr node != NULL) {
            printk(KERN ERR "binder: BINDER SET CONTEXT MGR already
set\n");
            ret = -EBUSY;
            goto err;
        }
        if (binder context mgr uid != -1) {
            if (binder context mgr uid != current->cred->euid) {
                printk(KERN ERR "binder: BINDER SET "
                    "CONTEXT MGR bad uid %d != %d\n",
                    current->cred->euid,
                    binder context mgr uid);
                ret = -EPERM;
                goto err;
            }
        } else
            binder context mgr uid = current->cred->euid;
        binder context mgr node = binder new node(proc, NULL, NULL);
        if (binder context mgr node == NULL) {
            ret = -ENOMEM;
            goto err;
        }
        binder context mgr node->local weak refs++;
        binder context mgr node->local strong refs++;
        binder context mgr node->has strong ref = 1;
        binder context mgr node->has weak ref = 1;
        break;
       ......
    default:
        ret = -EINVAL;
        goto err;
    }
    ret = 0;
err:
    if (thread)
        thread->looper &= ~BINDER LOOPER STATE NEED RETURN;
    mutex unlock(&binder lock);
    wait event interruptible(binder user error wait,
binder stop on user error < 2);
    if (ret && ret != -ERESTARTSYS)
        printk(KERN INFO "binder: %d:%d ioctl %x %lx returned %d\n",
proc->pid, current->pid, cmd, arg, ret);
    return ret;
}
```

在分析函数 binder_ioctl()之前，需要先弄明白如下两个数据结构：

(1) 结构体 binder_thread：表示一个线程，这里就是执行 binder_become_

 Android 的内存系统



context_manager()函数的线程。

```
struct binder thread {
    struct binder proc *proc;
    struct rb node rb node;
    int pid;
    int looper;
    struct binder transaction *transaction stack;
    struct list head todo;
    uint32 t return error; /* Write failed, return error code in read buf */
    uint32 t return error2; /* Write failed, return error code in read */
        /* buffer. Used when sending a reply to a dead process that */
        /* we are also waiting on */
    wait queue head t wait;
    struct binder stats stats;
};
```

在上述结构体中，proc 表示是这个线程所属的进程。结构体 binder_proc 中成员变量 thread 的类型是 rb_root，它表示一棵红黑树，把属于这个进程的所有线程都组织起来，结构体 binder_thread 的成员变量 rb_node 就是用来链入这棵红黑树的节点了。looper 成员变量表示线程的状态，它可以取下面的值：

```
enum {
    BINDER LOOPER STATE REGISTERED  = 0x01,
    BINDER LOOPER STATE ENTERED     = 0x02,
    BINDER LOOPER STATE EXITED      = 0x04,
    BINDER LOOPER STATE INVALID     = 0x08,
    BINDER LOOPER STATE WAITING     = 0x10,
    BINDER LOOPER STATE NEED RETURN = 0x20
};
```

至于其余的成员变量，transaction_stack 表示线程正在处理的事务，todo 表示发往该线程的数据列表，return_error 和 return_error2 表示操作结果返回码，wait 用来阻塞线程等待某个事件的发生，stats 用来保存一些统计信息。这些成员变量遇到的时候再分析它们的作用。

(2) 数据结构 binder_node：表示一个 binder 实体，定义如下。

```
struct binder node {
    int debug id;
    struct binder work work;
    union {
        struct rb node rb node;
        struct hlist node dead node;
    };
    struct binder proc *proc;
    struct hlist head refs;
    int internal strong refs;
    int local weak refs;
    int local strong refs;
    void __user *ptr;
```

```
    void   user *cookie;
    unsigned has strong ref : 1;
    unsigned pending strong ref : 1;
    unsigned has weak ref : 1;
    unsigned pending weak ref : 1;
    unsigned has async transaction : 1;
    unsigned accept fds : 1;
    int min priority : 8;
    struct list head async todo;
};
```

由此可见，rb_node 和 dead_node 组成了一个联合体，具体来说分为如下两种情形。

- 如果这个 Binder 实体还在正常使用，则使用 rb_node 来连入“proc->nodes”所表示的红黑树的节点，这棵红黑树用来组织属于这个进程的所有 Binder 实体。
- 如果这个 Binder 实体所属的进程已经销毁，而这个 Binder 实体又被其他进程所引用，则这个 Binder 实体通过 dead_node 进入到一个哈希表中去存放。proc 成员变量就是表示这个 Binder 实例所属于进程了。

refs 成员变量把所有引用了该 Binder 实体的 Binder 引用连接起来构成一个链表。internal_strong_refs、local_weak_refs 和 local_strong_refs 表示这个 Binder 实体的引用计数。ptr 和 cookie 成员变量分别表示这个 Binder 实体在用户空间的地址以及附加数据。其余的成员变量就不描述了，遇到的时候再分析。

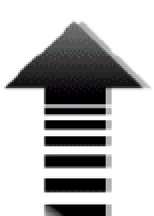
接下来回到函数 binder_ioctl()中，首先是通过 filp->private_data 获得 proc 变量，此处的函数 binder_mmap()是一样的，然后通过函数 binder_get_thread()获得线程信息，此函数的代码如下：

```
static struct binder thread *binder get thread(struct binder proc *proc)
{
    struct binder_thread *thread = NULL;
    struct rb node *parent = NULL;
    struct rb node **p = &proc->threads.rb node;

    while (*p) {
        parent = *p;
        thread = rb entry(parent, struct binder thread, rb node);

        if (current->pid < thread->pid)
            p = &(*p)->rb left;
        else if (current->pid > thread->pid)
            p = &(*p)->rb_right;
        else
            break;
    }
    if (*p == NULL) {
        thread = kzalloc(sizeof(*thread), GFP KERNEL);
        if (thread == NULL)
            return NULL;
        binder stats.obj created[BINDER STAT THREAD]++;
        thread->proc = proc;
```

 Android 的内存系统



```
        thread->pid = current->pid;
        init waitqueue head(&thread->wait);
        INIT LIST HEAD(&thread->todo);
        rb link node(&thread->rb node, parent, p);
        rb insert color(&thread->rb node, &proc->threads);
        thread->looper |= BINDER LOOPER STATE NEED RETURN;
        thread->return error = BR OK;
        thread->return error2 = BR OK;
    }
    return thread;
}
```

在上述代码中，把当前线程 current 的 pid 作为键值，在进程 proc->threads 表示的红黑树中进行查找，看是否已经为当前线程创建过了 binder_thread 信息。在这个场景下，由于当前线程是第一次进到这里，所以肯定找不到，即*p == NULL 成立，于是，就为当前线程创建一个线程上下文信息结构体 binder_thread，并初始化相应成员变量，并插入到 proc->threads 所表示的红黑树中去，下次要使用时就可以从 proc 中找到了。注意，这里的 thread->looper = BINDER_LOOPER_STATE_NEED_RETURN。

再回到函数 binder_ioctl()中，接下来会有两个全局变量 binder_context_mgr_node 和 binder_context_mgr_uid，定义如下：

```
static struct binder node *binder context mgr node;
static uid_t binder_context_mgr_uid = -1;
```

其中 binder_context_mgr_node 用来表示 Service Manager 实体，binder_context_mgr_uid 表示 Service Manager 守护进程的 uid。在这个场景下，由于当前线程是第一次进到这里，所以 binder_context_mgr_node 为 NULL，binder_context_mgr_uid 为-1，于是初始化 binder_context_mgr_uid 为 current->cred->euid，这样当前线程就成为 Binder 机制的守护进程了，并且通过 binder_new_node 为 Service Manager 创建 Binder 实体：

```
static struct binder node *
binder new node(struct binder proc *proc, void   user *ptr, void   user
*cookie)
{
    struct rb node **p = &proc->nodes.rb node;
    struct rb node *parent = NULL;
    struct binder node *node;
    while (*p) {
        parent = *p;
        node = rb_entry(parent, struct binder_node, rb_node);
        if (ptr < node->ptr)
            p = &(*p)->rb left;
        else if (ptr > node->ptr)
            p = &(*p)->rb right;
        else
            return NULL;
    }
    node = kzalloc(sizeof(*node), GFP KERNEL);
    if (node == NULL)
```

```
        return NULL;
    binder stats.obj created[BINDER STAT NODE]++;
    rb link node(&node->rb node, parent, p);
    rb insert color(&node->rb node, &proc->nodes);
    node->debug id = ++binder last id;
    node->proc = proc;
    node->ptr = ptr;
    node->cookie = cookie;
    node->work.type = BINDER WORK NODE;
    INIT LIST HEAD(&node->work.entry);
    INIT LIST HEAD(&node->async todo);
    if (binder debug mask & BINDER DEBUG INTERNAL REFS)
        printk(KERN INFO "binder: %d:%d node %d u%p c%p created\n",
              proc->pid, current->pid, node->debug id,
              node->ptr, node->cookie);
    return node;
}
```

在这里传进来的 ptr 和 cookie 都为 NULL。上述函数会首先检查 proc->nodes 红黑树中是否已经存在以 ptr 为键值的 node，如果已经存在则返回 NULL。在这个场景下，由于当前线程是第一次进入到这里，所以肯定不存在，于是就新建了一个 ptr 为 NULL 的 binder_node，并且初始化其他成员变量，并插入到 proc->nodes 红黑树中去。

当 binder_new_node 返回到函数 binder_ioctl()后，会把新建的 binder_node 指针保存在 binder_context_mgr_node 中，然后又初始化 binder_context_mgr_node 的引用计数值。这样执行 BINDER_SET_CONTEXT_MGR 命令完毕，在函数 binder_ioctl()返回之前执行下面的语句。

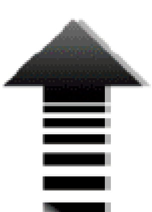
```
if (thread)
        thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
```

在执行 binder_get_thread 时，thread->looper = BINDER_LOOPER_STATE_NEED_RETURN，执行了这条语句后，thread->looper = 0。

再次回到文件 frameworks/base/cmds/servicemanager/service_manager.c 中的 main()函数，接下来需要调用函数 binder_loop()进入循环，等待 Client 发送请求。函数 binder_loop()定义在文件 frameworks/base/cmds/servicemanager/binder.c 中：

```
void binder loop(struct binder state *bs, binder handler func)
{
   int res;
   struct binder_write_read bwr;
   unsigned readbuf[32];
   bwr.write size = 0;
   bwr.write consumed = 0;
   bwr.write buffer = 0;

   readbuf[0] = BC ENTER LOOPER;
   binder write(bs, readbuf, sizeof(unsigned));
   for (;;) {
      bwr.read_size = sizeof(readbuf);
```

 Android 的内存系统


```
        bwr.read consumed = 0;
        bwr.read buffer = (unsigned) readbuf;
        res = ioctl(bs->fd, BINDER WRITE READ, &bwr);
        if (res < 0) {
            LOGE("binder loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }
        res = binder parse(bs, 0, readbuf, bwr.read consumed, func);
        if (res == 0) {
            LOGE("binder loop: unexpected reply?!\n");
            break;
        }
        if (res < 0) {
            LOGE("binder loop: io error %d %s\n", res, strerror(errno));
            break;
        }
    }
}
```

在上述代码中，首先通过函数 binder_write()执行 BC_ENTER_LOOPER 命令以告诉 Binder 驱动程序，Service Manager 马上要进入循环。在此还需要理解设备文件“/dev/binder”操作函数 ioctl 的操作码 BINDER_WRITE_READ，首先看其定义：

```
#define BINDER WRITE READ            IOWR('b', 1, struct
binder_write_read)
```

此 io 操作码有一个形式为 struct binder_write_read 的参数：

```
struct binder write read {
    signed long write size; /* bytes to write */
    signed long write consumed; /* bytes consumed by driver */
    unsigned long   write buffer;
    signed long read size;  /* bytes to read */
    signed long read consumed;  /* bytes consumed by driver */
    unsigned long   read buffer;
};
```

用户空间程序和 Binder 驱动程序交互时，大多数是通过 BINDER_WRITE_READ 命令实现的，write_buffer 和 read_buffer 所指向的数据结构还指定了具体要执行的操作，write_buffer 和 read_buffer 所指向的结构体是 binder_transaction_data，定义此结构体的代码如下。

```
struct binder transaction data {
    /* The first two are only used for bcTRANSACTION and brTRANSACTION,
     * identifying the target and contents of the transaction.
     */
    union {
        size t  handle; /* target descriptor of command transaction */
        void    *ptr;   /* target descriptor of return transaction */
    } target;
    void        *cookie;    /* target object cookie */
```

```
    unsigned int    code;       /* transaction command */

    /* General information about the transaction. */
    unsigned int    flags;
    pid_t       sender_pid;
    uid_t       sender_euid;
    size_t      data_size;  /* number of bytes of data */
    size_t      offsets_size;   /* number of bytes of offsets */

    /* If this transaction is inline, the data immediately
     * follows here; otherwise, it ends with a pointer to
     * the data buffer.
     */
    union {
        struct {
            /* transaction data */
            const void  *buffer;
            /* offsets from buffer to flat_binder_object structs */
            const void  *offsets;
        } ptr;
        uint8_t buf[8];
    } data;
};
```

到此为止，我们从源代码一步一步地分析完 Service Manager 是如何成为 Android 进程间通信(IPC)机制 Binder 守护进程的了。在接下来的内容中，简要总结 Service Manager 成为 Android 进程间通信(IPC)机制 Binder 守护进程的过程。

(1) 打开/dev/binder 文件：

```
open("/dev/binder", O_RDWR);
```

(2) 建立 128K 内存映射：

```
mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
```

(3) 通知 Binder 驱动程序它是守护进程：

```
binder_become_context_manager(bs);
```

(4) 进入循环等待请求的到来：

```
binder_loop(bs, svcmgr_handler);
```

在这个过程中，在 Binder 驱动程序中建立了一个 struct binder_proc 结构、一个 struct binder_thread 结构和一个 struct binder_node 结构，这样，Service Manager 就在 Android 系统的进程间通信机制 Binder 担负起守护进程的职责了。

5.2.3 分析 Server 和 Client 获得 Service Manager 的过程

作为守护进程，Service Manager 的职责是为 Server 和 Client 服务。那么，Server 和 Client 如何获得 Service Manager 接口，进而享受它提供的服务呢？在接下来的内容中，将

 Android 的内存系统



和大家一起分析 Server 和 Client 获得 Service Manager 的过程。

众所周知，Service Manager 在 Binder 机制中既充当守护进程的角色，同时它也充当着 Server 角色，但是它又与一般的 Server 不一样。对于普通的 Server 来说，Client 如果想要获得 Server 的远程接口，必须通过 Service Manager 远程接口提供的 getService 接口来获得，这本身就是一个使用 Binder 机制来进行进程间通信的过程。而对于 Service Manager 这个 Server 来说，Client 如果想要获得 Service Manager 远程接口，却不必通过进程间通信机制来获得，因为 Service Manager 远程接口是一个特殊的 Binder 引用，它的引用句柄一定是 0。

获取 Service Manager 远程接口的函数是 defaultServiceManager()，此函数声明在文件 frameworks/base/include/binder/IServiceManager.h 中，代码如下：

```
sp<IServiceManager> defaultServiceManager();
```

函数 defaultServiceManager()在 frameworks/base/libs/binder/IServiceManager.cpp 文件中实现：

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
    {
        AutoMutex  l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }
    return gDefaultServiceManager;
}
```

其中 gDefaultServiceManagerLock 和 gDefaultServiceManager 是全局变量，定义在文件 frameworks/base/libs/binder/Static.cpp 中：

```
Mutex gDefaultServiceManagerLock;
sp<IServiceManager> gDefaultServiceManager;
```

从上述函数可以看出，gDefaultServiceManager 是单例模式，在调用函数 defaultServiceManager()时，如果已经创建 gDefaultServiceManager，则直接返回，否则通过 interface_cast<IServiceManager>(ProcessState::self()->getContextObject(NULL))来创建一个，并保存在 gDefaultServiceManager 全局变量中。

在 Binder 机制中，类 BpServiceManager 继承了类 BpInterface<IServiceManager>，BpInterface 是一个模板类，它定义在文件 frameworks/base/include/binder/IInterface.h 中：

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase {
public:
    BpInterface(const sp<IBinder>& remote);
protected:
    virtual IBinder* onAsBinder();
};
```

类 IServiceManager 继承了类 IInterface，而类 IInterface 和类 BpRefBase 又分别继承了类 RefBase。在类 BpRefBase 中有一个名为 mRemote 的成员变量，它的类型是 IBinder*，实现类为 BpBinder，它表示一个 Binder 引用，引用句柄值保存在 BpBinder 类的 mHandle 成员变量中。类 BpBinder 通过类 IPCThreadState 来和 Binder 驱动程序并互，而 IPCThreadState 又通过它的成员变量 mProcess 来打开/dev/binder 设备文件，mProcess 成员变量的类型为 ProcessState。ProcessState 类打开设备/dev/binder 之后，将打开文件描述符保存在 mDriverFD 成员变量中，以供后续使用。

在理解了上述概念之后，接下来就可以继续分析创建 Service Manager 远程接口的过程了，我们的最终目的是要创建一个 BpServiceManager 实例，并且返回它的 IServiceManager 接口。下面是创建 Service Manager 远程接口的主要语句：

```
        gDefaultServiceManager = interface cast<IServiceManager>(
            ProcessState::self()->getContextObject(NULL));
```

上述代码虽然看似简短，但是暗藏玄机。首先是调用函数 ProcessState::self，函数 self()是 ProcessState 的静态成员函数，其作用是返回一个全局唯一的 ProcessState 实例变量，这就是单例模式，这个变量名为 gProcess。如果尚未创建 gProcess，就会执行创建操作，在 ProcessState 的构造函数中，会通过文件操作函数 open()打开设备文件“/dev/binder”，并且返回来的设备文件描述符保存在成员变量 mDriverFD 中。

接着调用函数 gProcess->getContextObject()获得一个句柄值为 0 的 Binder 引用，即 BpBinder，于是创建 Service Manager 远程接口的语句可以简化为下面的形式：

```
gDefaultServiceManager = interface cast<IServiceManager>(new
BpBinder(0));
```

再来看函数 interface_cast<IServiceManager>的具体实现，这是一个模板函数，定义在文件 framework/base/include/binder/IInterface.h 中：

```
template<typename INTERFACE>
inline sp<INTERFACE> interface cast(const sp<IBinder>& obj) {
    return INTERFACE::asInterface(obj);
}
```

这里的 INTERFACE 是 IServiceManager，调用了函数 IServiceManager::asInterface()。函数 IServiceManager::asInterface()是通过 DECLARE_META_INTERFACE(ServiceManager)宏在类 IServiceManager 中声明的，它位于文件 framework/base/include/binder/IServiceManager.h 中：

```
DECLARE_META_INTERFACE(ServiceManager);
```

展开后显示为：

```
#define DECLARE META INTERFACE(ServiceManager)                               \
    static const android::String16 descriptor;                             \
    static android::sp<IServiceManager> asInterface(                       \
    const android::sp<android::IBinder>& obj);                             \
    virtual const android::String16& getInterfaceDescriptor() const;      \
    IServiceManager();                                                     \
    virtual ~IServiceManager();
```

 Android 的内存系统



IServiceManager::asInterface 的实现是通过宏 IMPLEMENT_META_INTERFACE (ServiceManager, "android.os.IServiceManager")定义的，它位于文件 framework/base/libs/binder/IServiceManager.cpp 中：

```
IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager");
```

展开后即为：

```
#define IMPLEMENT META INTERFACE(ServiceManager,
"android.os.IServiceManager")                                                  \
    const android::String16
IServiceManager::descriptor("android.os.IServiceManager");                     \
    const android::String16&                                                   \
    IServiceManager::getInterfaceDescriptor() const {                          \
    return IServiceManager::descriptor;                                        \
    }                                                                          \
    android::sp<IServiceManager> IServiceManager::asInterface(                 \
    const android::sp<android::IBinder>& obj)                                  \
    {                                                                          \
    android::sp<IServiceManager> intr;                                         \
    if (obj != NULL) {                                                         \
    intr = static cast<IServiceManager*>(                                      \
    obj->queryLocalInterface(                                                  \
    IServiceManager::descriptor).get());                                       \
    if (intr == NULL) {                                                        \
    intr = new BpServiceManager(obj);                                          \
    }                                                                          \
    }                                                                          \
    return intr;                                                               \
    }                                                                          \
    IServiceManager::IServiceManager() { }                                     \
    IServiceManager::~IServiceManager() { }
```

IServiceManager::asInterface 的具体实现如下：

```
android::sp<IServiceManager> IServiceManager::asInterface(const
android::sp<android::IBinder>& obj)
{
    android::sp<IServiceManager> intr;

    if (obj != NULL) {
        intr = static cast<IServiceManager*>(
                 obj->queryLocalInterface(IServiceManager::descriptor).get());

        if (intr == NULL) {
            intr = new BpServiceManager(obj);
        }
    }
    return intr;
}
```

此处传进来的参数 obj 就是刚才创建的 new BpBinder(0)，类 BpBinder 中的成员函数 queryLocalInterface()继承自基类 IBinder，函数 IBinder::queryLocalInterface()位于文件 framework/base/libs/binder/Binder.cpp 中：

```
sp<IInterface>  IBinder::queryLocalInterface(const String16& descriptor)
{
   return NULL;
}
```

由此可见，在函数 IServiceManager::asInterface()中会调用下面的语句：

```
intr = new BpServiceManager(obj);
```

即为：

```
intr = new BpServiceManager(new BpBinder(0));
```

回到 defaultServiceManager()函数中，最终结果为：

```
gDefaultServiceManager = new BpServiceManager(new BpBinder(0));
```

这样，创建 Service Manager 远程接口完毕，它本质上是一个 BpServiceManager，包含了一个句柄值为 0 的 Binder 引用。

在 Android 系统的 Binder 机制中，Server 和 Client 拿到这个 Service Manager 远程接口之后怎么用呢？具体说明如下。

(1) 对于 Server 来说，就是调用接口 IServiceManager::addService 和 Binder 驱动程序进行交互，即调用 BpServiceManager::addService。而 BpServiceManager::addService 又会调用通过其基类 BpRefBase 的成员函数 remote()获得原先创建的 BpBinder 实例，接着调用成员函数 BpBinder::transact()。在函数 BpBinder::transact()中，又会调用成员函数 IPCThreadState::transact()，这里就是最终与 Binder 驱动程序交互的地方了。回忆一下前面的类图，IPCThreadState 有一个 PorcessState 类型的成中变量 mProcess，而 mProcess 有一个成员变量 mDriverFD，它是设备文件/dev/binder 的打开文件描述符，所以 IPCThreadState 相当于间接地拥有了设备文件“/dev/binder”的打开文件描述符，于是便可以与 Binder 驱动程序进行交互。

(2) 对于 Client 来说，就是调用 IServiceManager::getService 这个接口来和 Binder 驱动程序交互了。具体过程跟上述 Server 使用 Service Manager 的方法一样，在此不再介绍。

5.3　分析 Android 系统匿名共享内存 C++调用接口

在 Android 系统中，提供了独特的匿名共享内存子系统 Ashmem(Anonymous Shared Memory)，它以驱动程序的形式实现在内核空间中。Ashmem 有如下两个特点。

- 能够辅助内存管理系统来有效地管理不再使用的内存块。
- 通过 Binder 进程间通信机制来实现进程间的内存共享。

在本节的内容中，我们将通过实例来简要介绍 Android 系统匿名共享内存的使用方



法，使得我们对 Android 系统的匿名共享内存机制有一个感性的认识，为进一步学习它的源代码实现打下基础。

对于 Android 系统的匿名共享内存子系统来说，其主体是以驱动程序的形式实现在内核空间的，同时，在系统运行时库层和应用程序框架层提供了访问接口。其中在系统运行时库层提供了 C/C++调用接口，而在应用程序框架层提供了 Java 调用接口。在此我们将直接通过应用程序框架层提供的 Java 调用接口来说明匿名共享内存子系统 Ashmem 的使用方法，毕竟我们在 Android 开发应用程序时，是基于 Java 语言的。其实应用程序框架层的 Java 调用接口是通过 JNI 方法来调用系统运行时库层的 C/C++调用接口，最后进入到内核空间的 Ashmem 驱动程序去的。

下面举的例子是一个名为 Ashmem 的应用程序，它包含了一个 Server 端和一个 Client 端实现，其中 Server 端是以 Service 的形式实现的，在此 Service 里面，创建了一个匿名共享内存文件，而 Client 是一个 Activity，这个 Activity 通过 Binder 进程间通信机制获得前面这个 Service 创建的匿名共享内存文件的句柄，从而最终实现共享。在 Android 应用程序框架层中，提供了一个 MemoryFile 接口来封装匿名共享内存文件的创建和使用，它在文件 frameworks/base/core/java/android/os/MemoryFile.java 中实现。在接下来我们将分析 Server 端是如何通过类 MemoryFile 创建匿名共享内存文件的，并且分析 Client 是如何获得这个匿名共享内存文件的句柄的。

在类 MemoryFile 中提供了两种创建匿名共享内存的方法，在此我们将通过类 MemoryFile 的构造函数来看看这两种使用方法：

```
public class MemoryFile
{
    ......
    /**
    * Allocates a new ashmem region. The region is initially not purgable.
    *
    * @param name optional name for the file (can be null).
    * @param length of the memory file in bytes.
    * @throws IOException if the memory file could not be created.
    */
    public MemoryFile(String name, int length) throws IOException {
        mLength = length;
        mFD = native open(name, length);
        mAddress = native mmap(mFD, length, PROT READ | PROT WRITE);
        mOwnsRegion = true;
    }
    /**
    * Creates a reference to an existing memory file. Changes to the original file
    * will be available through this reference.
    * Calls to {@link #allowPurging(boolean)} on the returned MemoryFile
will fail.
    *
    * @param fd File descriptor for an existing memory file, as returned by
    *     {@link #getFileDescriptor()}. This file descriptor will be closed
    *     by {@link #close()}.
    * @param length Length of the memory file in bytes.
```

```
    * @param mode File mode. Currently only "r" for read-only access is
supported.
    * @throws NullPointerException if <code>fd</code> is null.
    * @throws IOException If <code>fd</code> does not refer to an
existing memory file,
    *      or if the file mode of the existing memory file is more restrictive
    *      than <code>mode</code>.
    *
    * @hide
    */
    public MemoryFile(FileDescriptor fd, int length, String mode) throws
IOException {
        if (fd == null) {
            throw new NullPointerException("File descriptor is null.");
        }
        if (!isMemoryFile(fd)) {
            throw new IllegalArgumentException("Not a memory file.");
        }
        mLength = length;
        mFD = fd;
        mAddress = native mmap(mFD, length, modeToProt(mode));
        mOwnsRegion = false;
    }
}
```

从上面的注释中可以看出这两个构造函数的使用方法。这两个构造函数的主要区别是第一个参数，具体说明如下。

(1) 第一种构造方法：以指定的字符串调用 JNI 方法 native_open 来创建一个匿名共享内存文件，从而得到一个文件描述符。接着以这个文件描述符为参数调用 JNI 方法 natvie_mmap，把这个匿名共享内存文件映射在进程空间中，然后就可以通过这个映射后得到的地址空间来直接访问内存数据了；

(2) 第二种构造方法：以指定的文件描述符来直接调用 JNI 方法 natvie_mmap，把这个匿名共享内存文件映射在进程空间中，然后进行访问，而这个文件描述符就必须要是一个匿名共享内存文件的文件描述符，这是通过一个内部函数 isMemoryFile 来验证的，而这个内部函数 isMemoryFile 也是通过 JNI 方法调用来进一步验证的。

这些 JNI 方法调用，最终都是通过系统运行时库层进入到内核空间的 Ashmem 驱动程序中去，不过在此我们无须关心这些 JNI 方法、系统运行库层调用以及 Ashmem 驱动程序的具体实现，而只需关注 MemoryFile 这个类的使用方法即可。

在我们举的例子中包含了一个 Server 端和一个 Client 端实现，其中，Server 端就是通过前面一个构造函数来创建一个匿名共享内存文件，接着 Client 端通过 Binder 进程间通信机制来向 Server 请求这个匿名共享内存的文件描述符，有了这个文件描述符之后，就可以通过后面一个构造函数来共享这个内存文件了。

因为涉及 Binder 进程间通信，我们首先定义好 Binder 进程间通信接口。首先在源代码工程的 packages/experimental 目录下创建一个应用程序工程目录 Ashmem，这样工程名称就是 Ashmem 了，它定义了一个路径为 shy.luo.ashmem 的 package，这个例子的源代码主要就是实现在这里了。下面，将会逐一介绍这个 package 里面的文件。

 Android 的内存系统

5.3.1 Java 程序

这里要用到的 Binder 进程间通信接口定义在文件 src/shy/luo/ashmem/IMemoryService.java 中：

```
package shy.luo.ashmem;

import android.util.Log;
import android.os.IInterface;
import android.os.Binder;
import android.os.IBinder;
import android.os.Parcel;
import android.os.ParcelFileDescriptor;
import android.os.RemoteException;

public interface IMemoryService extends IInterface {
    public static abstract class Stub extends Binder implements IMemoryService {
        private static final String DESCRIPTOR = "shy.luo.ashmem.IMemoryService";

        public Stub() {
            attachInterface(this, DESCRIPTOR);
        }

        public static IMemoryService asInterface(IBinder obj) {
            if (obj == null) {
                return null;
            }

           IInterface iin = (IInterface)obj.queryLocalInterface(DESCRIPTOR);
            if (iin != null && iin instanceof IMemoryService) {
                return (IMemoryService)iin;
            }

            return new IMemoryService.Stub.Proxy(obj);
        }

        public IBinder asBinder() {
            return this;
        }

        @Override
        public boolean onTransact(int code, Parcel data, Parcel reply,
int flags) throws android.os.RemoteException {
            switch (code) {
            case INTERFACE_TRANSACTION: {
                reply.writeString(DESCRIPTOR);
                return true;
            }
            case TRANSACTION_getFileDescriptor: {
```




```
                data.enforceInterface(DESCRIPTOR);

                ParcelFileDescriptor result = this.getFileDescriptor();

                reply.writeNoException();

                if (result != null) {
                    reply.writeInt(1);
                    result.writeToParcel(reply, 0);
                } else {
                    reply.writeInt(0);
                }

                return true;
            }
            case TRANSACTION setValue: {
                data.enforceInterface(DESCRIPTOR);

                int val = data.readInt();
                setValue(val);

                reply.writeNoException();

                return true;
            }
            }

            return super.onTransact(code, data, reply, flags);
        }

        private static class Proxy implements IMemoryService {
            private IBinder mRemote;

            Proxy(IBinder remote) {
                mRemote = remote;
            }

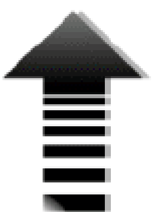
            public IBinder asBinder() {
                return mRemote;
            }

            public String getInterfaceDescriptor() {
                return DESCRIPTOR;
            }

            public ParcelFileDescriptor getFileDescriptor() throws
RemoteException {
                Parcel data = Parcel.obtain();
                Parcel reply = Parcel.obtain();

                ParcelFileDescriptor result;
```

 Android 的内存系统



```
                try {
                    data.writeInterfaceToken(DESCRIPTOR);

                    mRemote.transact(Stub.TRANSACTION getFileDescriptor,
data, reply, 0);

                    reply.readException();
                    if (0 != reply.readInt()) {
                        result =
ParcelFileDescriptor.CREATOR.createFromParcel(reply);
                    } else {
                        result = null;
                    }
                } finally {
                    reply.recycle();
                    data.recycle();
                }

                return result;
            }

            public void setValue(int val) throws RemoteException {
                Parcel data = Parcel.obtain();
                Parcel reply = Parcel.obtain();

                try {
                    data.writeInterfaceToken(DESCRIPTOR);
                    data.writeInt(val);

                   mRemote.transact(Stub.TRANSACTION setValue, data, reply, 0);

                    reply.readException();
                } finally {
                    reply.recycle();
                    data.recycle();
                }
            }
        }

        static final int TRANSACTION getFileDescriptor =
IBinder.FIRST CALL TRANSACTION + 0;
        static final int TRANSACTION setValue =
IBinder.FIRST CALL TRANSACTION + 1;

    }

    public ParcelFileDescriptor getFileDescriptor() throws
RemoteException;
    public void setValue(int val) throws RemoteException;
}
```

上述代码主要是定义了 IMemoryService 接口，在里面有如下两个调用接口：

```
    public ParcelFileDescriptor getFileDescriptor() throws
RemoteException;
    public void setValue(int val) throws RemoteException;
```

同时还分别定义了用于 Server 端实现的 IMemoryService.Stub 基类，还有用于 Client 端使用的代理 IMemoryService.Stub.Proxy 类。有了 Binder 进程间通信接口之后，接下来需要在 Server 端实现一个本地服务。此处 Server 端实现的本地服务名为 MemoryService，在文件 src/shy/luo/ashmem/MemoryService.java 中实现：

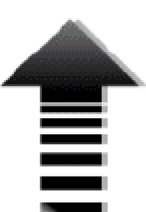
```
package shy.luo.ashmem;
import java.io.FileDescriptor;
import java.io.IOException;

import android.os.Parcel;
import android.os.MemoryFile;
import android.os.ParcelFileDescriptor;
import android.util.Log;

public class MemoryService extends IMemoryService.Stub {
   private final static String LOG TAG = "shy.luo.ashmem.MemoryService";
    private MemoryFile file = null;

    public MemoryService() {
        try {
                    file = new MemoryFile("Ashmem", 4);
                    setValue(0);
            }
            catch(IOException ex) {
                    Log.i(LOG TAG, "Failed to create memory file.");
                    ex.printStackTrace();
            }
    }
    public ParcelFileDescriptor getFileDescriptor() {
        Log.i(LOG TAG, "Get File Descriptor.");
        ParcelFileDescriptor pfd = null;
        try {
            pfd = file.getParcelFileDescriptor();
        } catch(IOException ex) {
            Log.i(LOG TAG, "Failed to get file descriptor.");
            ex.printStackTrace();
        }
        return pfd;
    }

    public void setValue(int val) {
        if(file == null) {
            return;
        }
```



```
        byte[] buffer = new byte[4];
        buffer[0] = (byte)((val >>> 24) & 0xFF);
        buffer[1] = (byte)((val >>> 16) & 0xFF);
        buffer[2] = (byte)((val >>> 8) & 0xFF);
        buffer[3] = (byte)(val & 0xFF);

        try {
            file.writeBytes(buffer, 0, 0, 4);
            Log.i(LOG_TAG, "Set value " + val + " to memory file. ");
        }
        catch(IOException ex) {
            Log.i(LOG_TAG, "Failed to write bytes to memory file.");
            ex.printStackTrace();
        }
    }
}
```

读者在此需要注意，这里的类 MemoryService 实现了类 IMemoryService.Stub，表示这是一个 Binder 服务的本地实现。在构造函数中，通过指定文件名和文件大小来创建了一个匿名共享内存文件，即创建 MemoryFile 的一个实例，并保存在类成员变量 file 中。这个匿名共享内存文件名为“Ashmem”，大小为 4 个节字，刚好容纳一个整数。我们这里举的例子就是要说明如果创建一个匿名共享内存来在两个进程间实现共享一个整数。其实在实际应用中，可以根据需要创建合适大小的共享内存来共享有意义的数据。

这里还实现了 IMemoryService.Stub 的两个接口：getFileDescriptor 和 setVal，其中一个用来获取匿名共享内存文件的文件描述符，一个来往匿名共享内存文件中写入一个整数。接口 getFileDescriptor 的返回值是一个 ParcelFileDescriptor。在 Java 中，用类 FileDescriptor 来表示一个文件描述符，而 ParcelFileDescriptor 是用来序列化 FileDescriptor 的，以便在进程间调用时传输。

定义好本地服务好后，需要定义一个 Server 来启动这个服务了。这里定义的 Server 在文件 src/shy/luo/ashmem/Server.java 中实现：

```
package shy.luo.ashmem;
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import android.os.ServiceManager;

public class Server extends Service {
    private final static String LOG_TAG = "shy.luo.ashmem.Server";

    private MemoryService memoryService = null;
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    @Override
    public void onCreate() {
```

```
    Log.i(LOG TAG, "Create Memory Service...");
    memoryService = new MemoryService();
       try {
          ServiceManager.addService("AnonymousSharedMemory",
memoryService);
          Log.i(LOG TAG, "Succeed to add memory service.");
       } catch (RuntimeException ex) {
          Log.i(LOG TAG, "Failed to add Memory Service.");
          ex.printStackTrace();
       }
   }
   @Override
   public void onStart(Intent intent, int startId) {
      Log.i(LOG TAG, "Start Memory Service.");
   }
   @Override
   public void onDestroy() {
    Log.i(LOG TAG, "Destroy Memory Service.");
   }
}
```

此 Server 继承了 Android 系统应用程序框架层提供的类 Service，当它被启动时，运行在一个独立的进程中。当这个 Server 被启动时，其 onCreate()函数就会被调用，然后它就通过 ServiceManager 的接口 addService 来添加 MemoryService：

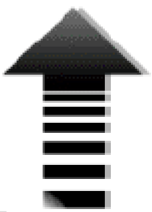
```
   memoryService = new MemoryService();
   try {
      ServiceManager.addService("AnonymousSharedMemory", memoryService);
      Log.i(LOG TAG, "Succeed to add memory service.");
   } catch (RuntimeException ex) {
      Log.i(LOG_TAG, "Failed to add Memory Service.");
      ex.printStackTrace();
   }
```

这样当此 Server 成功启动后，Client 就可以通过 ServiceManager 的 getService 接口来获取这个 MemoryService。

接下来开始看 Client 端的实现。Client 端是一个 Activity，在文件 src/shy/luo/ashmem/Client.java 中实现：

```
package shy.luo.ashmem;
import java.io.FileDescriptor;
import java.io.IOException;
import shy.luo.ashmem.R;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.os.MemoryFile;
import android.os.ParcelFileDescriptor;
import android.os.ServiceManager;
import android.os.RemoteException;
import android.util.Log;
```

 Android 的内存系统



```
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
public class Client extends Activity implements OnClickListener {
    private final static String LOG TAG = "shy.luo.ashmem.Client";

    IMemoryService memoryService = null;
    MemoryFile memoryFile = null;

    private EditText valueText = null;
    private Button readButton = null;
    private Button writeButton = null;
    private Button clearButton = null;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        IMemoryService ms = getMemoryService();
        if(ms == null) {
            startService(new Intent("shy.luo.ashmem.server"));
        } else {
            Log.i(LOG TAG, "Memory Service has started.");
        }
        valueText = (EditText)findViewById(R.id.edit value);
        readButton = (Button)findViewById(R.id.button read);
        writeButton = (Button)findViewById(R.id.button write);
        clearButton = (Button)findViewById(R.id.button clear);
        readButton.setOnClickListener(this);
        writeButton.setOnClickListener(this);
        clearButton.setOnClickListener(this);

        Log.i(LOG TAG, "Client Activity Created.");
    }
    @Override
    public void onResume() {
        super.onResume();
        Log.i(LOG TAG, "Client Activity Resumed.");
    }
    @Override
    public void onPause() {
        super.onPause();
        Log.i(LOG TAG, "Client Activity Paused.");
    }
    @Override
    public void onClick(View v) {
        if(v.equals(readButton)) {
            int val = 0;

            MemoryFile mf = getMemoryFile();
            if(mf != null) {
```

```
                try {
                    byte[] buffer = new byte[4];
                    mf.readBytes(buffer, 0, 0, 4);

                    val = (buffer[0] << 24) | ((buffer[1] & 0xFF) << 16)
| ((buffer[2] & 0xFF) << 8) | (buffer[3] & 0xFF);
                } catch(IOException ex) {
                    Log.i(LOG_TAG, "Failed to read bytes from memory file.");
                    ex.printStackTrace();
                }
            }
            String text = String.valueOf(val);
            valueText.setText(text);
        } else if(v.equals(writeButton)) {
            String text = valueText.getText().toString();
            int val = Integer.parseInt(text);

            IMemoryService ms = getMemoryService();
            if(ms != null) {
                try {
                    ms.setValue(val);
                } catch(RemoteException ex) {
                    Log.i(LOG_TAG, "Failed to set value to memory service.");
                    ex.printStackTrace();
                }
            }
        } else if(v.equals(clearButton)) {
            String text = "";
            valueText.setText(text);
        }
    }
    private IMemoryService getMemoryService() {
        if(memoryService != null) {
            return memoryService;
        }
        memoryService = IMemoryService.Stub.asInterface(

    ServiceManager.getService("AnonymousSharedMemory"));
        Log.i(LOG_TAG, memoryService != null ? "Succeed to get memeory
service." : "Failed to get memory service.");

        return memoryService;
    }

    private MemoryFile getMemoryFile() {
        if(memoryFile != null) {
            return memoryFile;
        }

        IMemoryService ms = getMemoryService();
        if(ms != null) {
```

 Android 的内存系统


```
                try {
                    ParcelFileDescriptor pfd = ms.getFileDescriptor();
                    if(pfd == null) {
                        Log.i(LOG_TAG, "Failed to get memory file descriptor.");
                        return null;
                    }
                    try {
                        FileDescriptor fd = pfd.getFileDescriptor();
                        if(fd == null) {
                           Log.i(LOG_TAG, "Failed to get memeory file descriptor.");
                            return null;
                        }
                        memoryFile = new MemoryFile(fd, 4, "r");
                    } catch(IOException ex) {
                        Log.i(LOG_TAG, "Failed to create memory file.");
                        ex.printStackTrace();
                    }
                } catch(RemoteException ex) {
                    Log.i(LOG_TAG, "Failed to get file descriptor from memory
service.");
                    ex.printStackTrace();
                }
            }
            return memoryFile;
        }
}
```

在 Client 端的界面主要包含了三个按钮 Read、Write 和 Clear，以及一个用于显示内容的文本框。当此 Activity 在 onCreate 时，会通过 startService 接口来启动我们前面定义的 Server 进程。当调用 startService 时，需要指定要启动服务的名称。

内部函数 getMemoryService()用来获取 IMemoryService。如果是第一次调用该函数，则会通过 ServiceManager 的接口 getService 来获得这个 IMemoryService 接口，然后保存在类成员变量 memoryService 中，以后再调用这个函数时，就可以直接返回 memoryService。

内部函数 getMemoryFile()用来从 MemoryService 中获得匿名共享内存文件的描述符。如果是第一次调用该函数，则会通过 IMemoryService 的接口 getFileDescriptor 来获得 MemoryService 中的匿名共享内存文件的描述符，然后用这个文件描述符来创建一个 MemoryFile 实例，并保存在类成员变量 memoryFile 中，以后再调用这个函数时，就可以直接返回 memoryFile 了。

这样当有了 memoryService 和 memoryFile 后，就可以在 Client 端访问 Server 端创建的匿名共享内存了。单击 Read 按钮时，就通过 memoryFile 的 readBytes 接口把共享内存中的整数读出来，并显示在文本框中；单击 Write 按钮时，就通过 memoryService 这个代理类的 setVal 接口来调用 MemoryService 的本地实现类的 setVal 服务，从而把文本框中的数值写到 Server 端创建的匿名共享内存中去；单击 Clear 按钮时，就会清空文本框的内容。这样，我们就可以通过 Read 和 Write 按钮来验证我们是否在 Client 和 Server 两个进程中实现内存共享了。

5.3.2 相关程序

现在，我们再来看看 Client 界面的配置文件，在文件 res/layout/main.xml 中定义的代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout width="fill parent"
   android:layout height="fill parent"
   >
   <LinearLayout
    android:layout width="fill parent"
    android:layout height="wrap content"
    android:orientation="vertical"
    android:gravity="center">
    <TextView
           android:layout width="wrap content"
           android:layout height="wrap content"
        android:text="@string/value">
    </TextView>
    <EditText
           android:layout width="fill parent"
        android:layout height="wrap content"
        android:id="@+id/edit value"
        android:hint="@string/hint">
    </EditText>
   </LinearLayout>
    <LinearLayout
    android:layout width="fill parent"
    android:layout height="wrap content"
    android:orientation="horizontal"
    android:gravity="center">
    <Button
        android:id="@+id/button read"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:text="@string/read">
    </Button>
    <Button
        android:id="@+id/button write"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:text="@string/write">
    </Button>
    <Button
        android:id="@+id/button clear"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:text="@string/clear">
```

 Android 的内存系统

```
    </Button>
   </LinearLayout>
</LinearLayout>
```

相关的字符串定义在 res/values/strings.xml 文件中：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app name">Ashmem</string>
   <string name="value">Value</string>
   <string name="hint">Please input a value...</string>
   <string name="read">Read</string>
   <string name="write">Write</string>
   <string name="clear">Clear</string>
</resources>
```

接下来看程序描述文件 AndroidManifest.xml 的相关配置，位于 Ashmem 目录下：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
     package="shy.luo.ashmem"
     android:sharedUserId="android.uid.system"
     android:versionCode="1"
     android:versionName="1.0">
   <application android:icon="@drawable/icon"
android:label="@string/app name">
      <activity android:name=".Client"
               android:label="@string/app name">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
       <service
           android:enabled="true"
           android:name=".Server"
           android:process=".Server" >
           <intent-filter>
             <action android:name="shy.luo.ashmem.server"/>
             <category android:name="android.intent.category.DEFAULT"/>
         </intent-filter>
       </service>
   </application>
</manifest>
```

在此可以看到，下面的配置项把服务名称“shy.luo.ashmem.server”和本地服务类 Server 关联了起来：

```
   <service
    android:enabled="true"
    android:name=".Server"
    android:process=".Server" >
    <intent-filter>
```

```
            <action android:name="shy.luo.ashmem.server"/>
            <category android:name="android.intent.category.DEFAULT"/>
        </intent-filter>
    </service>
```

这样就可以通过 startService(new Intent("shy.luo.ashmem.server"))启动这个 Server。但是在 Android 中启动服务是需要权限的，所以下面的一行配置获取了启动服务需要的相应权限：

```
android:sharedUserId="android.uid.system"
```

接下来看工程的编译脚本文件 Android.mk，它位于 Ashmem 目录下：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES += $(call all-subdir-java-files)
LOCAL_PACKAGE_NAME := Ashmem
LOCAL_CERTIFICATE := platform
include $(BUILD_PACKAGE)
```

上述倒数第二行很重要，因为我们需要在程序中启动 Service，所以要配置这一行，并且要把源代码工程放在 Android 源代码平台中进行编译。

这样，整个例子的源代码实现就介绍完了，接下来就要编译了。执行以下命令进行编译和打包：

```
USER-NAME@MACHINE-NAME:~/Android$ mmm packages/experimental/Ashmem
USER-NAME@MACHINE-NAME:~/Android$ make snod
```

这样打包好的 Android 系统镜像文件 system.img 会包含我们前面创建的 Ashmem 应用程序了。接下来就可以通过模拟器来运行我们的例子，执行以下命令启动模拟器：

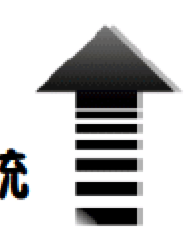
```
USER-NAME@MACHINE-NAME:~/Android$ emulator
```

启动模拟器后可以在 Home Screen 中看到 Ashmem 应用程序图标，如图 5-2 所示。



图 5-2　Ashmem 应用程序图标

 Android 的内存系统



单击 Ashmem 图标，启动 Ashmem 应用程序，界面如图 5-3 所示。

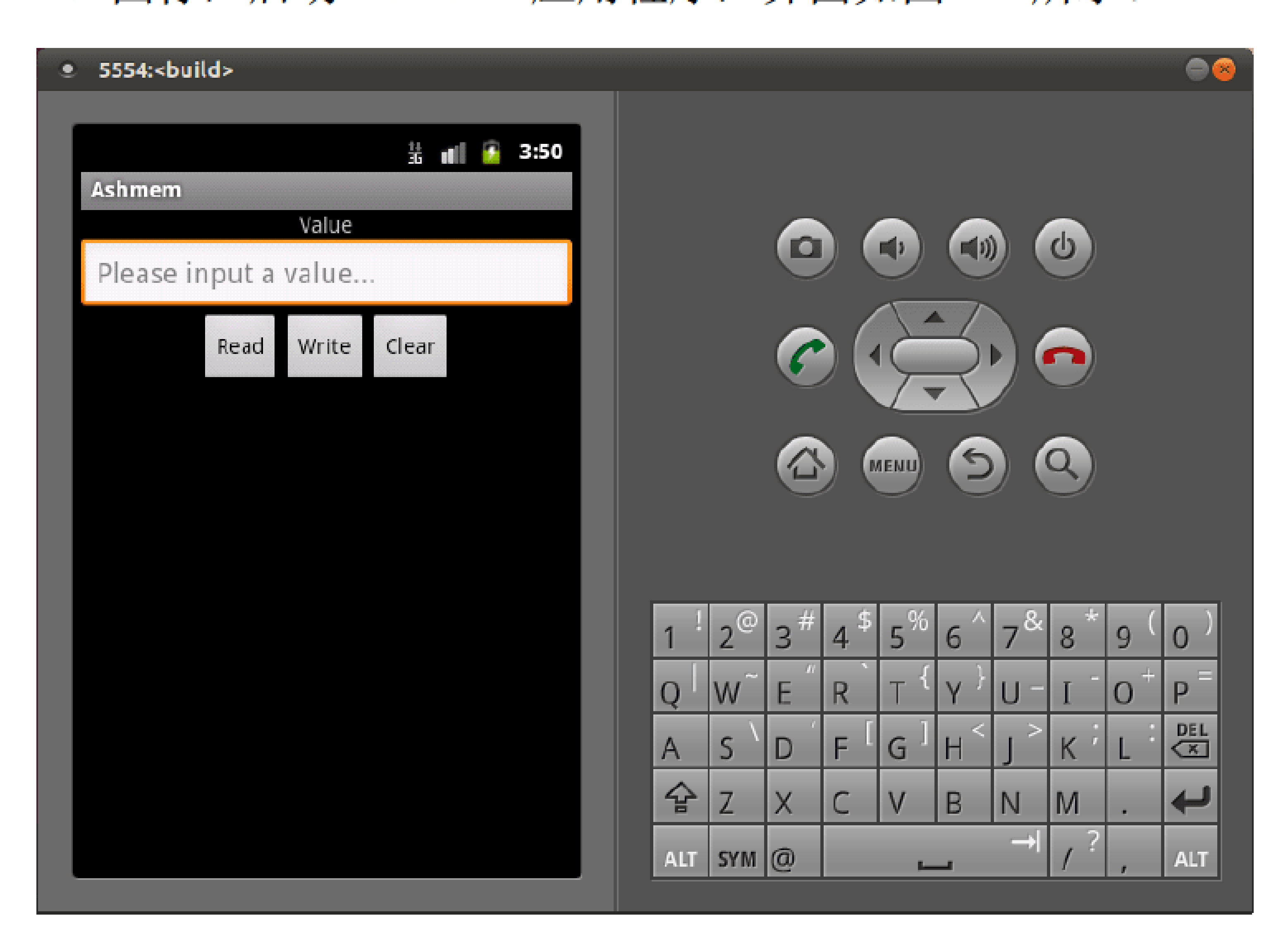


图 5-3　启动后的界面

此时就可以验证程序的功能了，看看是否实现了在两个进程中通过使用 Android 系统的匿名共享内存机制来共享内存数据的功能。

5.4　Android 中的垃圾回收

垃圾回收机制比较重要，能够达到节约内存的目的，并最终实现提高手机的处理效率。本节将简单讲解 Android 系统中的垃圾回收机制。

5.4.1　sp 和 wp 简析

在 Android 系统中，sp 和 wp 被称为智能指针(android refbase 类(sp 和 wp))。其实 sp 和 wp 就是 Android 为其 C++实现的自动垃圾回收机制。如果具体到内部实现，sp 和 wp 实际上只是一个实现垃圾回收功能的接口而已，比如说对*，->的重载，是为了其看起来跟真正的指针一样，而真正实现垃圾回收的是 refbase 这个基类。这部分代码位于如下文件中：

```
/frameworks/base/include/utils/RefBase.h
```

在此所有的类都会虚继承于 refbase 类，因为它实现了达到 Android 垃圾回收所需要的所有 function，因此实际上所有的对象声明出来以后都具备了自动释放自己的能力。也就是说实际上智能指针就是我们的对象本身，它会维持一个对本身强引用和弱引用的计数，一旦强引用计数为 0 它就会释放掉自己。

(1) sp

sp 实际上不是 smart pointer 的缩写，而是 strong pointer，它实际上内部就包含了一个

指向对象的指针而已。我们可以简单看看 sp 的一个构造函数：

```
template< typename T>
sp< T>::sp(T* other)
: m ptr(other)
{
if (other) other->incStrong(this);
}
```

比如说我们声明一个对象：

```
sp< CameraHardwareInterface> hardware(new CameraHal());
```

实际上 sp 指针对本身没有进行什么操作，就是一个指针的赋值，包含了一个指向对象的指针，但是对象会对对象本身增加一个强引用计数，这个 incStrong 的实现就在 refbase 类里面。新 new 出来一个 CameraHal 对象，将它的值给 sp< CameraHardwareInterface>的时候，它的强引用计数就会从 0 变为 1。因此每次将对象赋值给一个 sp 指针的时候，对象的强引用计数都会加 1，下面我们再看看 sp 的析构函数：

```
template< typename T>
sp< T>::~sp()
{
if (m ptr) m ptr->decStrong(this);
}
```

实际上每次删除一个 sp 对象的时候，sp 指针指向的对象的强引用计数就会减 1，当对象的强引用计数为 0 的时候，这个对象就会被自动释放掉。

(2) wp

我们再看 wp，wp 就是 weak pointer 的缩写，弱引用指针的原理，就是为了应用 Android 垃圾回收来减少对那些胖子对象对内存的占用，我们首先来看 wp 的一个构造函数：

```
wp< T>::wp(T* other)
: m ptr(other)
{
if (other) m refs = other->createWeak(this);
}
```

它和 sp 一样，实际上也就是仅仅对指针进行了赋值而已，对象本身会增加一个对自身的弱引用计数，同时 wp 还包含一个 m_ref 指针，这个指针主要是用来将 wp 升级为 sp 时使用的：

```
template< typename T>
sp< T> wp< T>::promote() const
{
return sp< T>(m ptr, m refs);
}
template< typename T>
sp< T>::sp(T* p, weakref type* refs)
: m_ptr((p && refs->attemptIncStrong(this)) ? p : 0)
```

 Android 的内存系统



```
{
}
```

实际上我们对 wp 指针唯一能做的就是将 wp 指针升级为一个 sp 指针，然后判断是否升级成功，如果成功说明对象依旧存在，如果失败说明对象已经被释放掉了。wp 指针现在在单例中使用很多，确保 mhardware 对象只有一个，比如：

```
wp< CameraHardwareInterface> CameraHardwareStub::singleton;
sp< CameraHardwareInterface> CameraHal::createInstance()
{
LOG FUNCTION NAME
if (singleton != 0) {
sp< CameraHardwareInterface> hardware = singleton.promote();
if (hardware != 0) {
return hardware;
}
}
sp< CameraHardwareInterface> hardware(new CameraHal()); //强引用加 1
singleton = hardware;//弱引用加 1
return hardware;//赋值构造函数，强引用加 1
}
//hardware 被删除，强引用减 1
```

5.4.2 详解智能指针(android refbase 类(sp 和 wp))

在 Android 的源代码中，经常会看到形如 sp<xxx>、wp<xxx>形式的类型定义，这其实是 Android 中的智能指针。智能指针是 C++中的一个概念，通过基于引用计数的方法，解决对象自动释放的问题。在 C++编程中，有两个很让人头痛的问题：一是忘记释放动态申请的对象，从而造成内存泄漏；二是对象在一个地方释放后，又在别的地方被使用，从而引起内存访问错误。程序员往往需要花费很大精力进行精心设计，以避免这些问题的出现。在使用智能指针后，动态申请的内存将会被自动释放(有点类似 Java 的垃圾回收)，不需要再使用 delete 来释放对象，也不需要考虑一个对象是否已经在其他地方被释放了，从而使程序编写工作减轻不少，而程序的稳定性大大提高。

Android 的智能指针相关的源代码在如下两个文件中：

```
frameworks/base/include/utils/RefBase.h
frameworks/base/libs/utils/RefBase.cpp
```

涉及的类以及类之间的关系如图 5-4 所示。

Android 中定义了两种智能指针类型：一种是强指针 sp(strong pointer)，一种是弱指针(weak pointer)。其实称为强引用和弱引用更合适一些。强指针与一般意义的智能指针概念相同，通过引用计数来记录有多少使用者在使用一个对象，如果所有使用者都放弃了对该对象的引用，则该对象将被自动销毁。

弱指针也指向一个对象，但是弱指针仅仅记录该对象的地址，不能通过弱指针来访问该对象，也就是说不能通过弱指针来调用对象的成员函数或访问对象的成员变量。要想访问弱指针所指向的对象，需首先将弱指针升级为强指针(通过 wp 类所提供的 promote()方

法)。弱指针所指向的对象是有可能在其他地方被销毁的。如果对象已经被销毁，wp 的 promote()方法将返回空指针，这样就能避免出现地址访问错的情况。

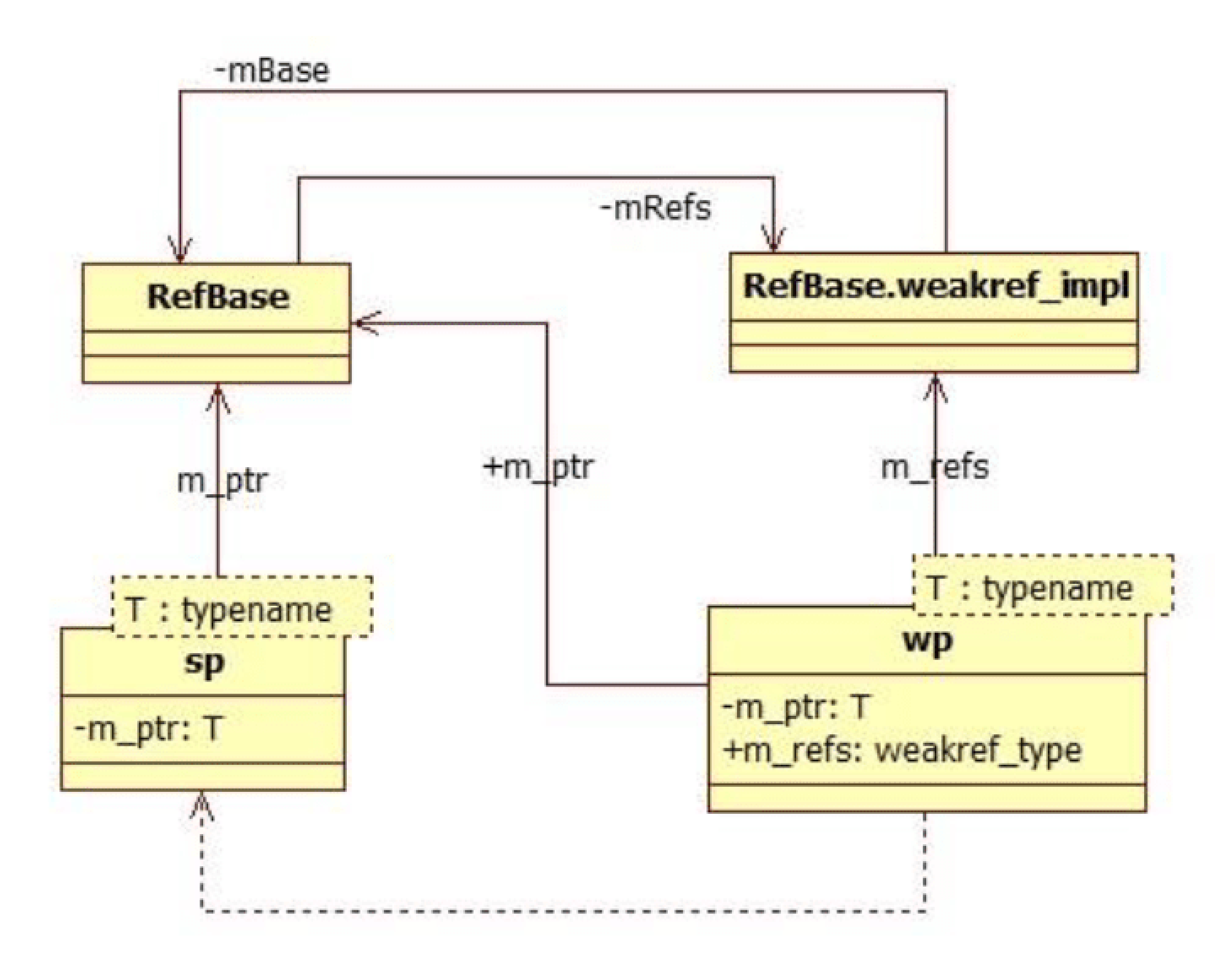


图 5-4　智能指针相关类的关系

究竟指针是怎么做到这一点的呢？其实一点也不复杂，原因就在于每一个可以被智能指针引用的对象，都同时被附加了另外一个 weakref_impl 类型的对象，这个对象中负责记录对象的强指针引用计数和弱指针引用计数。这个对象是智能指针的实现内部使用的，智能指针的使用者看不到这个对象。弱指针操作的就是这个对象，只有当强引用计数和弱引用计数都为 0 时，这个对象才会被销毁。

接下来开始分析到底该怎么使用智能指针。假设现在有一个类 MyClass，如果要使用智能指针来引用这个类的对象，那么这个类需满足下列两个前提条件：

(1) 这个类是基类 RefBase 的子类或间接子类；

(2) 这个类必须定义虚构造函数，即它的构造函数需要这样定义：

```
virtual ~MyClass();
```

满足了上述条件的类后就可以定义智能指针，定义方法和普通指针类似。比如普通指针是这样定义：

```
MyClass* p_obj;
```

智能指针是这样定义：

```
sp<MyClass> p_obj;
```

注意不要定义成 sp<MyClass>* p_obj。初学者很容易犯这种错误，这样实际上相当于定义了一个指针的指针。尽管在语法上没有问题，但是最好永远不要使用这样的定义。

定义了一个智能指针的变量，就可以像普通指针那样使用它，包括赋值、访问对象成员、作为函数的返回值、作为函数的参数等。例如：

```
p_obj = new MyClass(); // 注意不要写成 p_obj = new sp<MyClass>
```

 Android 的内存系统

```
sp<MyClass> p_obj2 = p_obj;
p_obj->func();
p_obj = create_obj();
some_func(p_obj);
```

注意不要试图 delete(删除)一个智能指针，即 delete p_obj。不要担心对象的销毁问题，智能指针的最大作用就是自动销毁不再使用的对象。不需要再使用一个对象后，直接将指针赋值为 NULL 即可：

```
p_obj = NULL;
```

上面说的都是强指针，弱指针的定义方法和强指针类似，但是不能通过弱指针来访问对象的成员。下面是弱指针的示例：

```
wp<MyClass> wp_obj = new MyClass();
p_obj = wp_obj.promote(); // 升级为强指针。不过这里要用.而不是->，真是有负其指针
之名啊
wp_obj = NULL;
```

由此可见，智能指针用起来很方便，在一般情况下最好使用智能指针来代替普通指针。但是需要知道一个智能指针其实是一个对象，而不是一个真正的指针，因此其运行效率是远远比不上普通指针的。所以在对运行效率敏感的地方，最好还是不要使用智能指针为好。

第 6 章

Android 内存优化

在第 5 章中已经讲解了 Android 系统内存的运作原理和机制。通过这些内容，为我们本章将要讲解的内存优化知识做好了铺垫。希望读者专心学习本章中关于内存优化的基本内容，为步入本书后面高级知识的学习打下基础。

6.1 Android 内存优化的作用

Android 系统以其开源免费、界面优美，得到了大众的青睐。各种为其量身定做的软件层出不穷。其中不乏系统优化类软件，但是所谓的内存优化真的有用吗？

Android 应用程序使用 Java 作为开发语言。aapt 工具把编译后的 Java 代码连同其他应用程序需要的数据和资源文件一起打包到一个 Android 包文件中，这个文件使用“.apk”格式作为扩展名，它是分发应用程序并安装到移动设备的媒介，用户只需下载并安装此文件到他们的设备。单一.apk 文件中的所有代码被认为是一个应用程序。

从很多方面来看，每个 Android 应用程序都存在于它自己的世界之中：

- 默认情况下，每个应用程序均运行于它自己的 Linux 进程中。当应用程序中的任意代码开始执行时，Android 启动一个进程，而当不再需要此进程而其他应用程序又需要系统资源时，则关闭这个进程。
- 每个进程都运行于自己的 Java 虚拟机(VM)中。所以应用程序代码实际上与其他应用程序的代码是隔绝的。
- 默认情况下，每个应用程序均被赋予一个唯一的 Linux 用户 ID，并加以权限设置，使得应用程序的文件仅对这个用户、这个应用程序可见。当然，也有其他的方法使得这些文件同样能为别的应用程序所访问。

使两个应用程序共有同一个用户 ID 是可行的，这种情况下他们可以看到彼此的文件。从系统资源维护的角度来看，拥有同一个 ID 的应用程序也将在运行时使用同一个 Linux 进程，以及同一个虚拟机。

谈到 Android，自然离不开 Java 语言，其比传统的 C/C++等编程语言的一个明显优点就是解决了内存泄漏的问题。这个时候，内存优化便被推到了前台。

Java 程序的内存分配与回收都是由 JRE 在后台自动进行的。JRE 会负责回收那些不再使用的内存，也就是大家听说的内存回收机制(Garbage Collection，也叫 GC)。

Java 的堆内存是一个运行时(Runtime)数据区，用以保存对象，Java 虚拟机堆内存中存储着正在运行的应用程序所建立的所有对象，这些对象不需要程序通过代码来显式地释放。垃圾回收是一种动态存储管理技术，它自动释放不再被程序引用的对象，按照特定的垃圾回收算法来实现内存资源的自动回收功能。

总之一句话，Java 的内存都是由 JVM(Java 虚拟机)控制的。可能有些人可能会说，既然 Java 会自动回收内存，那为什么我的手机显示占用内存达到 70%，甚至 80%呢？先不说这个原因，我就问一句内存占用 80%的时候，和你用所谓的内存优化软件优化之后，手机运行速度有改变吗？至于内存占用较高的问题，就是 JVM 的一个缺点了(毕竟人无完人)。JVM 必须跟踪程序中的有用对象，才可以确定哪些对象是无用的，并最终释放这些无用对象，所以需要额外占用一部分内存。

6.2　查看 Android 内存和 CPU 使用情况

本节将首先简单介绍查看 Android 系统单个进程内存和 CPU 的使用情况，具体来说有 4 种方法。

6.2.1　利用 Android API 函数查看

利用 ActivityManager 可以查看可以用内存，例如下面的代码：

```
ActivityManager.MemoryInfo outInfo = new ActivityManager.MemoryInfo();
am.getMemoryInfo(outInfo);
```

在上述代码中，outInfo.availMem 表示可用的空闲内存。

另外，可以使用 Android.os.Debug 来查询 PSS、VSS 和 USS 等单个进程使用内存信息。例如下面的代码：

```
MemoryInfo[] memoryInfoArray = am.getProcessMemoryInfo(pids);
MemoryInfo pidMemoryInfo=memoryInfoArray[0];
pidMemoryInfo.getTotalPrivateDirty();
getTotalPrivateDirty()
Return total private dirty memory usage in kB. USS
getTotalPss()
Return total PSS memory usage in kB.
PSS
getTotalSharedDirty()
Return total shared dirty memory usage in kB. RSS
```

6.2.2　直接对 Android 文件进行解析查询

大家都知道，Android 实际上是一个 Linux 的衍生系统，虽然 Google 曾经在 Linux kernel 之上增加了一些专用的内存分配驱动(ashmen,pmem 这两部分会随着这个议题的深入来讨论，这里暂时忽略)，但是关于一般应用程序的内存使用还是采用 Linux 的传统方法，因此我们通过 Linux 的/proc 文件系统的 meminfo 来分析这个系统的内存使用情况更客观。

之所以这么说，是因为通过这种方法可以绕开繁琐的 dalvik 实现机制，以系统的层面来分析。具体来说，在“/proc/cpuinfo 系统”中保存了 CPU 等的多种信息，而在/proc/meminfo 中保存了系统内存的使用信息。

例如在/proc/meminfo 中存在如下信息：

```
MemTotal: 16344972 kB
MemFree: 13634064 kB
Buffers: 3656 kB
Cached: 1195708 kB
```

我们查看机器内存时，会发现 MemFree 的值很小。这主要是因为，在 Linux 中有这么

一种思想，内存不用白不用，因此它尽可能地 cache 和 buffer 一些数据，以方便下次使用。但实际上这些内存也是可以立刻拿来使用的，所以：

```
空闲内存=free+buffers+cached=total-used
```

通过读取文件/proc/meminfo 的信息获取 Memory 的总量。通过 ActivityManager.getMemoryInfo(ActivityManager.MemoryInfo)可以获取当前的可用 Memory 量。

6.2.3 通过 Runtime 类实现

通过 Android 系统提供的 Runtime 类，然后执行 adb 命令(top,procrank,ps...等命令)即可实现查询功能。通过对执行结果的标准控制台输出进行解析，这样就大大地扩展了 Android 的查询功能。例如下面的演示：

```
final Process m process = Runtime.getRuntime().exec("/system/bin/top -n 1");
final StringBuilder sbread = new StringBuilder();
BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(m process.getInputStream()), 8192);

# procrank
Runtime.getRuntime().exec("/system/xbin/procrank");
```

内存耗用分别用 VSS、RSS、PSS 或 USS 来表示，具体说明如下：

- VSS：是 Virtual Set Size 的缩写，表示虚拟耗用内存(包含共享库占用的内存)；
- RSS：是 Resident Set Size 的缩写，实际使用物理内存(包含共享库占用的内存)；
- PSS：是 Proportional Set Size 的缩写，实际使用的物理内存(比例分配共享库占用的内存)；
- USS：是 Unique Set Size 的缩写，进程独自占用的物理内存(不包含共享库占用的内存)。

一般来说，内存占用大小有如下规律：

```
VSS >= RSS >= PSS >= USS
```

例如，下面是读取 Android 设备的内存数据(USS、PSS 和 RSS)的演示代码：

```
final ActivityManager am = (ActivityManager)
getSystemService(ACTIVITY SERVICE);
Android.os.Debug.MemoryInfo[] memoryInfoArray =
am.getProcessMemoryInfo(new int[]{android.os.Process.myPid()});
```

其中类 MemoryInfo 提供了 API 接口帮助我们获取内存数据，获取各种数据的对应函数如下。

- 函数 getTotalPrivateDirty()：获取 USS 数据；
- 函数 getTotalSharedDirty()：获取 RSS 数据；
- 函数 getTotalPss()：获取 PSS 数据。

 Android 内存优化



6.2.4 使用 DDMS 工具获取

有很多读者在拿到 Android 设备以后，可能会试着通过 Google 提供的工具来获得系统的内存使用情况。Google 提供了一个工具叫 DDMS，通过此工具可以获取内存的使用状况。

DDMS 的全称是 Dalvik Debug Monitor Service，它为我们提供例如：为测试设备截屏、针对特定的进程查看正在运行的线程以及堆信息、Logcat、广播状态信息、模拟电话呼叫、接收 SMS、虚拟地理坐标等。

1. 如何启动 DDMS

DDMS 工具存放在“SDK－tools/”路径下，启动 DDMS 的方法如下。

(1) 直接双击 ddms.bat 运行。

(2) 在 Eclipes 调试程序的过程中启动 DDMS，在 Eclipes 中的界面如图 6-1 所示。然后选择 Other 命令，打开如图 6-2 所示的对话框。



图 6-1 Eclipes 中的界面

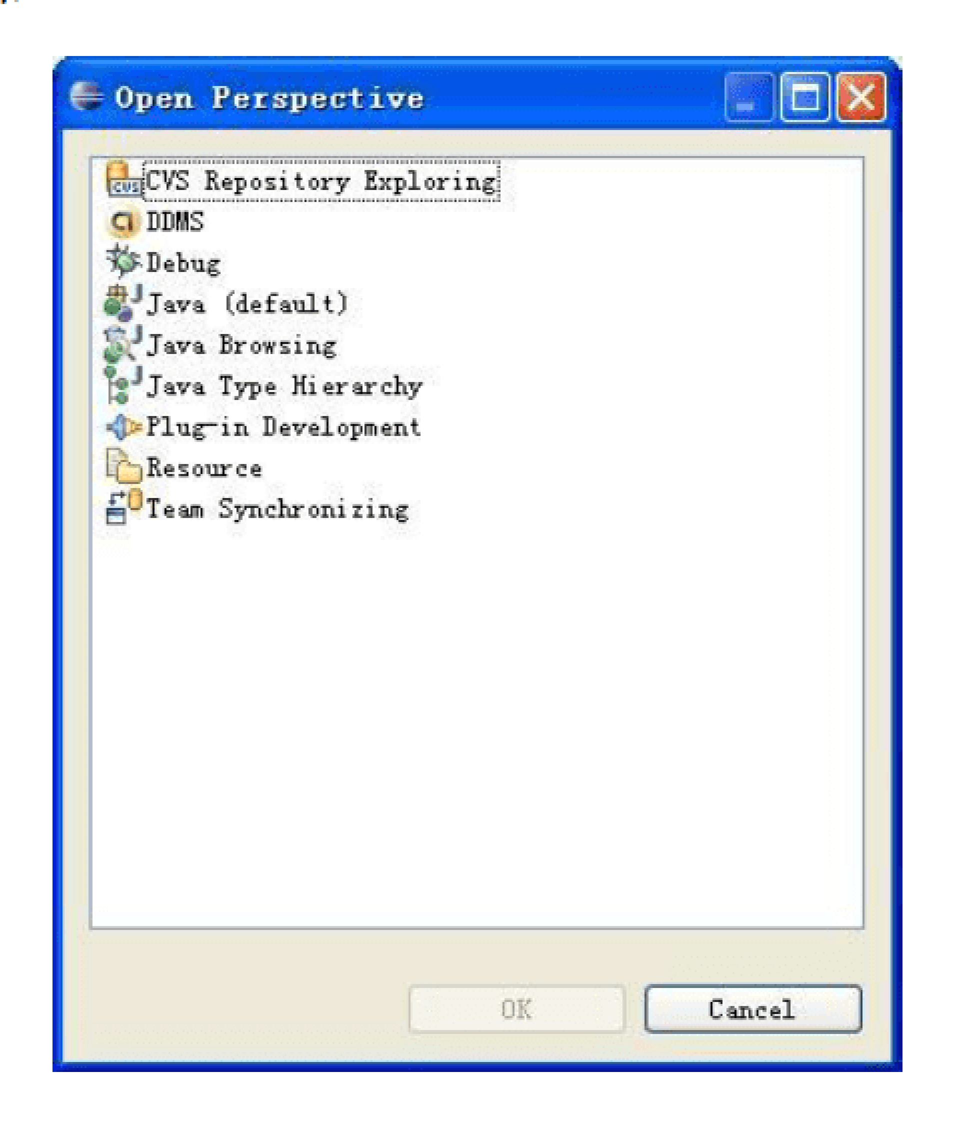


图 6-2 Open Perspective 对话框

此时双击 DDMS 就可以启动了。DDMS 对 Emulator 和外接测试机有同等效用。如果系统检测到它们(VM)同时运行，那么 DDMS 将会默认指向 Emulator。以上两种启动后的操作有些不一样，建议分别尝试一下。

2. DDMS 的工作原理

DDMS 将搭建起 IDE 与测试终端(Emulator 或者 connected device)的链接，它们应用各自独立的端口监听调试器的信息，DDMS 可以实时监测到测试终端的连接情况。当有新的测试终端连接后，DDMS 将捕捉到终端的 ID(即运行进程)，如图 6-3 所示。并通过 adb 建立调试器，从而实现发送指令到测试终端的目的。

DDMS 监听第一个终端 App 进程的端口为 8600，APP 进程将分配 8601，如果有更多终端或者更多 APP 进程将按照这个顺序依次类推。DDMS 通过 8700 端口(“base port”)

接收所有终端的指令。

Devices

Name			
system_process	572		8600
android.process.acore	617		8601 / ...
com.android.phone	619		8602
android.process.media	663		8611
com.android.alarmclock	686		8613
com.example.android.snake	740		8616
com.android.inputmethod.lat	902		8603

图 6-3　捕捉到终端的 ID

在 GUI 的左上角可以看到标签为“Devices”的面板，这里可以查看到所有与 DDMS 连接的终端的详细信息，以及每个终端正在运行的 APP 进程，每个进程最右边相对应的是与调试器链接的端口。因为 Android 是基于 Linux 内核开发的操作平台，同时也保留了 Linux 中特有的进程 ID，它介于进程名和端口号之间，如图 6-4 所示。

Devices

Name			
system_process	572		8600
android.process.acore	617		8601 / ...
com.android.phone	619		8602
android.process.media	663		8611
com.android.alarmclock	686		8613
com.example.android.snake	740		8616
com.android.inputmethod.lat	902		8603

图 6-4　连接终端的信息

在面板的右上角有一排很重要的按键他们分别是 Debug the selected process、Update Threads、Update Heap、Stop Process 和 ScreenShot。

3. Emulator Control

通过这个面板的一些功能可以非常容易的使测试终端模拟真实手机所具备的一些交互功能，比如：接听电话，根据选项模拟各种不同网络情况，模拟接受 SMS 消息和发送虚拟地址坐标用于测试 GPS 功能等，如图 6-5 所示。

图 6-5 中各个选项的具体说明如下。

- Telephony Status：通过选项模拟语音质量以及信号连接模式。
- Telephony Actions：模拟电话接听和发送 SMS 到测试终端。
- Location Control：模拟地理坐标或者模拟动态的路线坐标变化并显示预设的地理标识，可以通过以下 3 种方式。
  - Manual：手动为终端发送二维经纬坐标。
  - GPX：通过 GPX 文件导入序列动态变化地理坐标，从而模拟行进中 GPS 变化的数值。
  - KML：通过 KML 文件导入独特的地理标识，并以动态形式根据变化的地理坐标显示在测试终端。

Android 内存优化



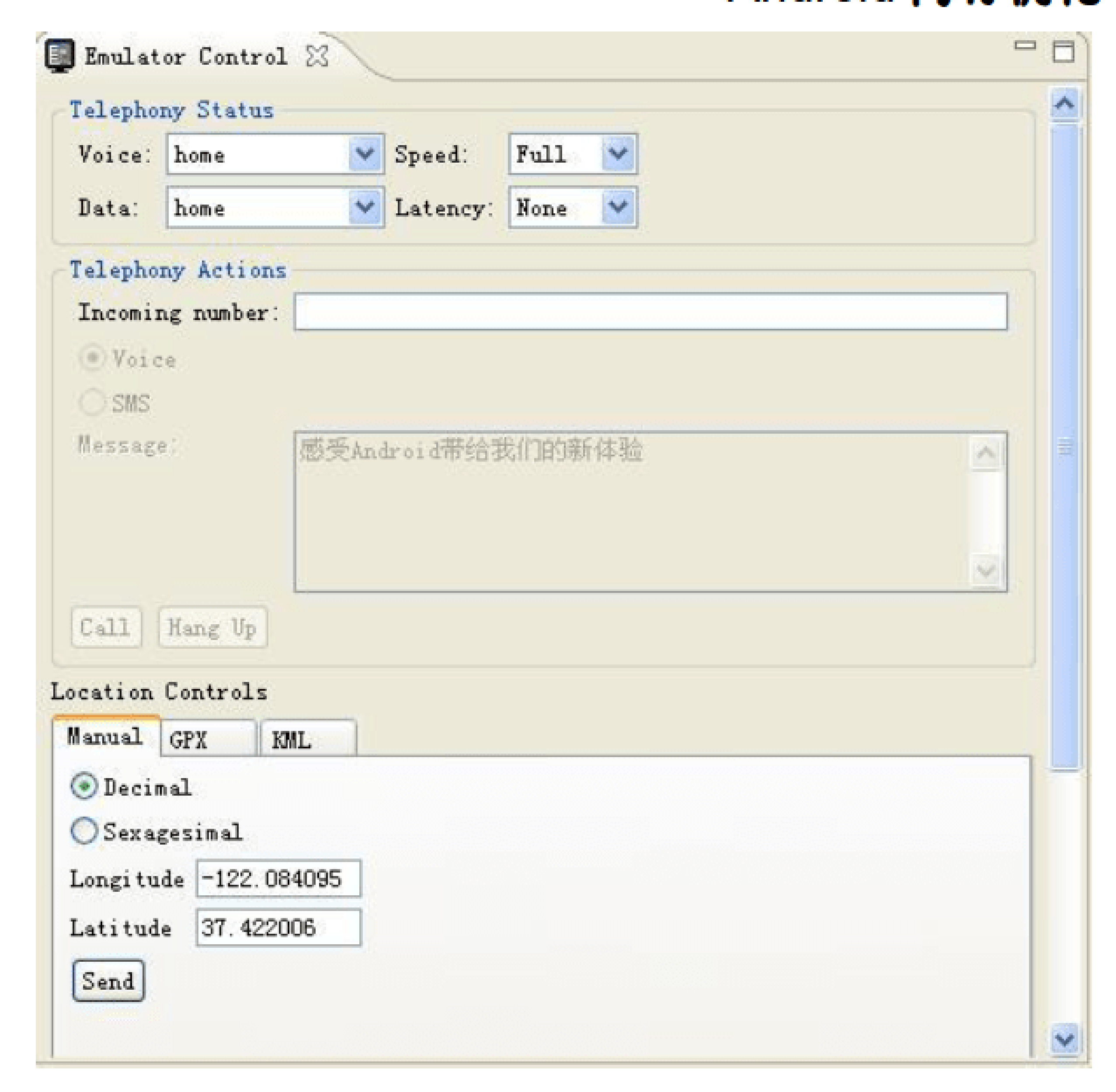


图 6-5　Emulator Control 面板

4. Threads、Heap、File Explorer

Threads、Heap、File Explorer 属于同一面板，例如 Heap 的界面如图 6-6 所示。

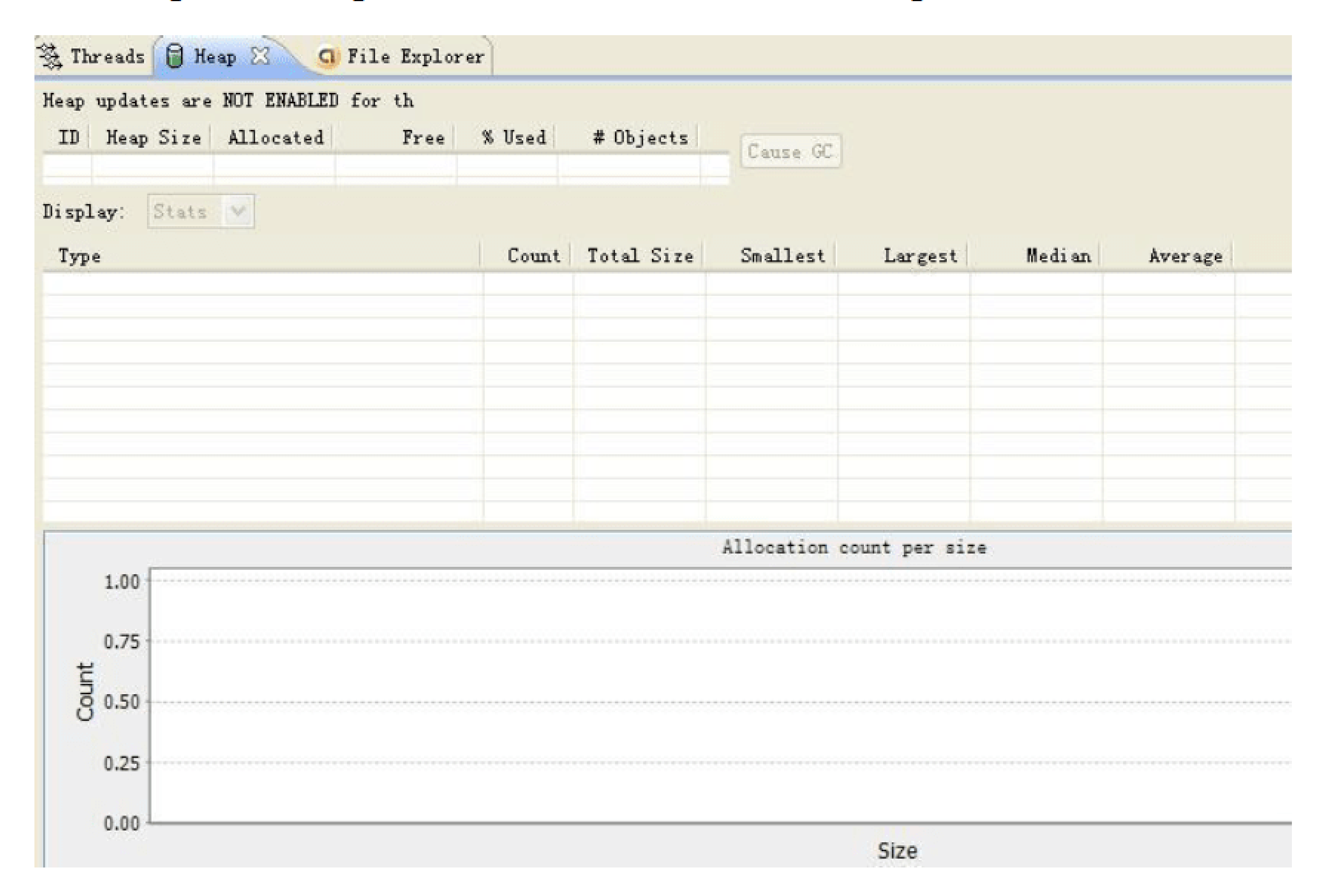


图 6-6　Heap 的界面

通过 File Explorer 可以查看 Android 模拟器中的文件，可以很方便地导入/导出文件。

5. Locate、Console

Locate 用于显示输出的调试信息，Console 是 Android 模拟器输出的信息，加载程序等信息。界面如图 6-7 所示。

```
LogCat  Console
Android
[2009-09-11 13:42:57 - Snake] Installing Snake.apk...
[2009-09-11 13:43:24 - Snake] Application already exists. Attempting to re-install instead...
[2009-09-11 13:43:37 - Snake] Success!
[2009-09-11 13:43:37 - Snake] Starting activity com.example.android.snake.Snake on device
[2009-09-11 13:43:41 - Snake] ActivityManager: Starting: Intent { comp={com.example.android.snake/com.example
[2009-09-11 13:57:04 - Emulator] emulator: emulator window was out of view and was recentred
[2009-09-11 13:57:04 - Emulator]
[2009-09-11 13:57:11 - Snake] ------------------------------
[2009-09-11 13:57:11 - Snake] Android Launch!
[2009-09-11 13:57:11 - Snake] adb is running normally.
[2009-09-11 13:57:11 - Snake] Performing com.example.android.snake.Snake activity launch
[2009-09-11 13:57:11 - Snake] Automatic Target Mode: launching new emulator with compatible AVD 'moandroid'
[2009-09-11 13:57:11 - Snake] Launching a new emulator with Virtual Device 'moandroid'
```

图 6-7　Locate 和 Console 面板

6．使用 DDMS 获取内存数据

在 DDMS 中有一个很不错的内存监测工具 Heap，使用 Heap 可以监测应用进程使用内存情况，具体操作步骤如下：

(1) 启动 Eclipse 后，切换到 DDMS 透视图，并确认 Devices 视图、Heap 视图都是打开的；

(2) 将手机通过 USB 链接至电脑，链接时需要确认手机是处于“USB 调试”模式，而不是作为“Mass Storage”；

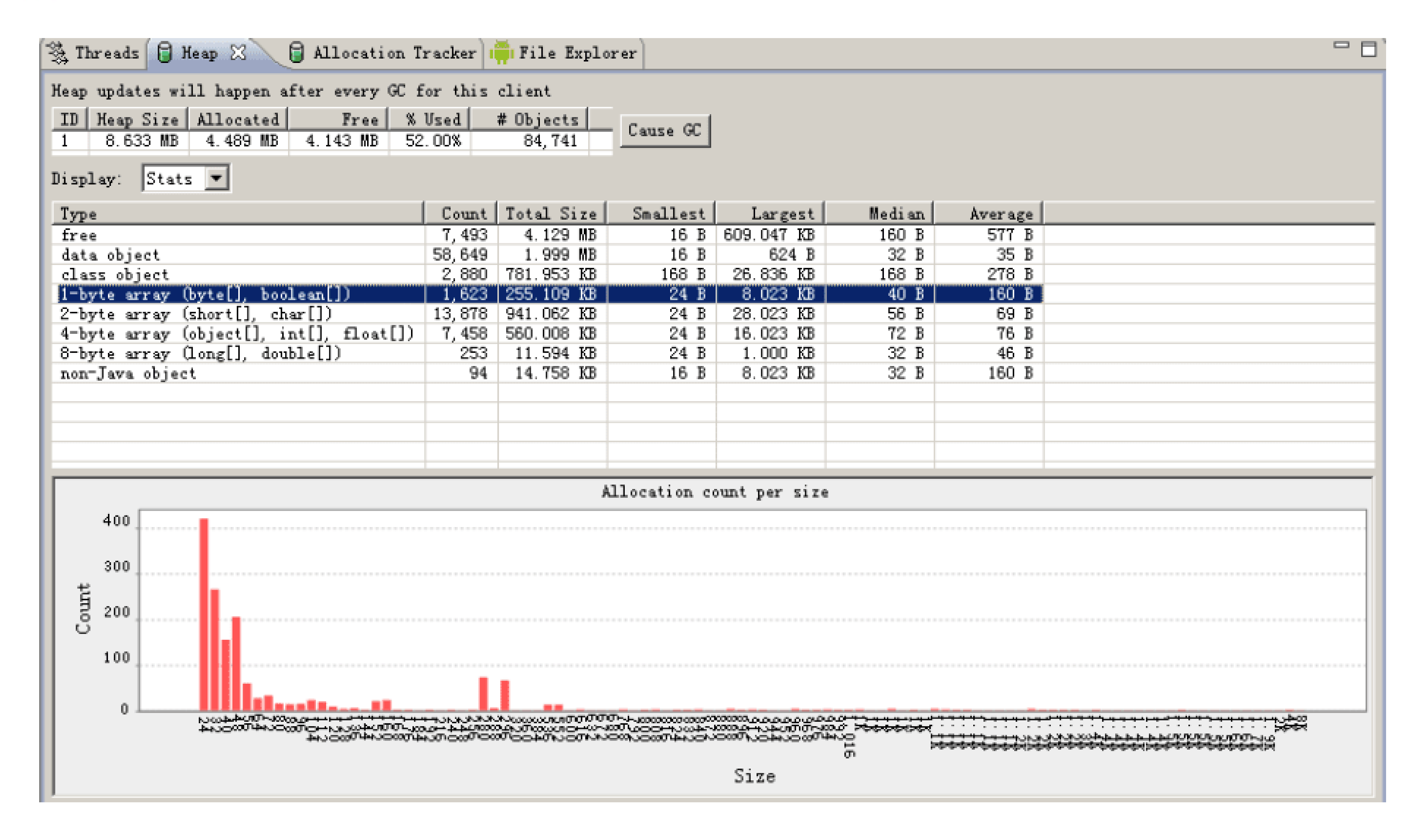
(3) 链接成功后，在 DDMS 的 Devices 视图中将会显示手机设备的序列号，以及设备中正在运行的部分进程信息；

(4) 单击选中想要监测的进程，比如 system_process 进程；

(5) 单击选中 Devices 视图界面中最上方一排图标中的 Update Heap 图标；

(6) 单击 Heap 视图中的 Cause GC 按钮；

(7) 此时在 Heap 视图中就会看到当前选中的进程的内存使用量的详细情况，例如如图 6-8 所示。



ID	Heap Size	Allocated	Free	% Used	# Objects
1	8.633 MB	4.489 MB	4.143 MB	52.00%	84,741

Type	Count	Total Size	Smallest	Largest	Median	Average
free	7,493	4.129 MB	16 B	609.047 KB	160 B	577 B
data object	58,649	1.999 MB	16 B	624 B	32 B	35 B
class object	2,880	781.953 KB	168 B	26.836 KB	168 B	278 B
1-byte array (byte[], boolean[])	1,623	255.109 KB	24 B	8.023 KB	40 B	160 B
2-byte array (short[], char[])	13,878	941.062 KB	24 B	28.023 KB	56 B	69 B
4-byte array (object[], int[], float[])	7,458	560.008 KB	24 B	16.023 KB	72 B	76 B
8-byte array (long[], double[])	253	11.594 KB	24 B	1.000 KB	32 B	46 B
non-Java object	94	14.758 KB	16 B	8.023 KB	32 B	160 B

图 6-8　内存使用情况


 Android 内存优化

在图 6-8 中列出了现在系统的一些进程和使用情况。其中系统随时可以用的两项内存是 Free 和 Buffers，因为笔者设置的系统只有 128M 的内存，所以看上去这部分可用内存已经很少了。笔者在此系统试着跑很占内存的游戏等应用程序的时候，并没有发现内存不足的问题。鉴于这个原因，认为这张图并不能反映出我要得到的系统内存资源信息，因此我只能从另一个角度去分析它。

接下来我们用 6.2.2 节的方法进行获取，请看/proc/meminfo 数据的截图，如图 6-9 所示。

```
ricky@ricky-laptop:~$ adb shell
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
# cat /proc/meminfo
MemTotal:         107756 kB
MemFree:            1984 kB
Buffers:            2464 kB
Cached:            51988 kB
SwapCached:            0 kB
Active:            43476 kB
Inactive:          49976 kB
Active(anon):      34096 kB
Inactive(anon):     5080 kB
Active(file):       9380 kB
Inactive(file):    44896 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:         39012 kB
Mapped:            26772 kB
Slab:               5164 kB
SReclaimable:       2424 kB
SUnreclaim:         2740 kB
PageTables:         3536 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:       53876 kB
Committed_AS:    1069292 kB
VmallocTotal:     385024 kB
VmallocUsed:       33996 kB
VmallocChunk:     344068 kB
#
```

图 6-9 /proc/meminfo 数据的截图

在图 6-9 所示的截图中，对于 Linux 系统来说，可以立即使用的内存是：

```
MemFree+Buffers+Cache=556436kB
```

系统总共可用的内存为：

```
MemTotal = 107756kB
```

通过运算我们可以发现实际上系统目前还有 52%的内存处于空闲状态，和我们从 DDMS 中拿到的图差很多。或者说 Google 隐藏了 cache，没有给我我想要的东西。

由此可见，Android 系统为了加快系统的运行速度会在系统允许的情况下，大量的使用内存作为应用程序的 cache。而当系统内存紧张的时候，会首先释放 cache 的内存，这也就是我依旧能跑占内存比较大的游戏的原因。由此可以总结道，如果想得到每个 Android APP 的内存比例可以用 DDMS 来得到，如果想判断系统内存更详细的信息可以用 Linux 的 proc/meminfo。

6.2.5 其他方法

除了本节前面介绍的 4 种方法外，接下来还介绍几种其他查看内存的方法。

1. Running Services 方式

我们可以通过手机上 Running Services 的 Activity 查看内存，依次点击 Setting->Applications->Running services 来实现。

2. 使用 ActivityManager 的 getMemoryInfo(ActivityManager.MemoryInfo outInfo)

函数 ActivityManager.getMemoryInfo()的主要功能是，得到当前系统剩余内存的及判断是否处于低内存运行。例如下面的演示代码：

```
    private void displayBriefMemory() {
        final ActivityManager activityManager = (ActivityManager)
getSystemService(ACTIVITY SERVICE);
        ActivityManager.MemoryInfo info = new ActivityManager.MemoryInfo();
        activityManager.getMemoryInfo(info);
        Log.i(tag,"系统剩余内存:"+(info.availMem >> 10)+"k");
        Log.i(tag,"系统是否处于低内存运行："+info.lowMemory);
        Log.i(tag,"当系统剩余内存低于"+info.threshold+"时就看成低内存运行");
    }
```

函数 ActivityManager.getMemoryInfo()是用 ActivityManager.MemoryInfo 返回结果，而不是 Debug.MemoryInfo。

ActivityManager.MemoryInfo 只有如下三个 Field：

- availMem：表示系统剩余内存；
- lowMemory：是 boolean 值，表示系统是否处于低内存运行；
- hreshold：它表示当系统剩余内存低于好多时就看成低内存运行。

3. 在代码中使用 Debug 的 getMemoryInfo(Debug.MemoryInfo memoryInfo)或 ActivityManager 的 MemoryInfo[] getProcessMemoryInfo(int[] pids)

该方式能够得到比较详细地 MemoryInfo 所描述的内存使用情况，数据单位是 KB。Android 和 Linux 一样有大量内存在进程之间进程共享。某个进程准确的使用好多内存实际上是很难统计的。

因为有 Paging out to Disk(换页)的存在，所以如果把所有映射到进程的内存相加，它可能大于内存的实际物理大小。此方法 MemoryInfo 的 Field 说明如下。

- dalvik：是指 dalvik 所使用的内存。
- native：是被 Native(本地)堆使用的内存，应该指使用 C\C++在堆上分配的内存。
- other：是指除 dalvik 和 native 使用的内存。但是具体是指什么呢？至少包括在 C\C++分配的非堆内存，比如分配在栈上的内存。
- private：是指私有的、非共享的。
- share：是指共享的内存。
- PSS：实际使用的物理内存(比例分配共享库占用的内存)。



- Pss：它是把共享内存根据一定比例分摊到共享它的各个进程来计算所得到进程使用内存。网上又说是比例分配共享库占用的内存，那么至于这里的共享是否只是库的共享，还是不清楚。
- PrivateDirty：它是指非共享的，又不能换页出去(can not be paged to disk)的内存的大小。比如 Linux 为了提高分配内存速度而缓冲的小对象，即使你的进程结束，该内存也不会释放掉，它只是又重新回到缓冲中而已。
- SharedDirty：是指共享的，又不能换页出去(Can not be paged to Disk)的内存的大小。比如 Linux 为了提高分配内存速度而缓冲的小对象，即使所有共享它的进程结束，该内存也不会释放掉，它只是又重新回到缓冲中而已。

MemoryInfo 所描述的内存使用情况都可以通过命令 adb shell "dumpsys meminfo %curProcessName%" 得到。如果想在代码中同时得到多个进程的内存使用或非本进程的内存使用情况请使用 ActivityManager 的 MemoryInfo[] getProcessMemoryInfo(int[] pids)，否则 Debug 的 getMemoryInfo(Debug.MemoryInfo memoryInfo)就可以了。

我们可以通过 ActivityManager 的 List<ActivityManager.RunningAppProcessInfo> getRunningAppProcesses()得到当前所有运行的进程信息。ActivityManager. RunningAppProcessInfo 中就有进程的 id，名字以及该进程包括的所有 apk 包名列表等。

4. 使用 Debug 的方法

这里的 Debug 的方法是指 getNativeHeapSize()、getNativeHeapAllocatedSize()、getNativeHeapFreeSize()共三个方法。该方式只能得到 Native 堆的内存大概情况，数据单位为字节。

- static long getNativeHeapAllocatedSize()：返回的是当前进程 navtive 堆中已使用的内存大小。
- static long getNativeHeapFreeSize()：返回的是当前进程 navtive 堆中已经剩余的内存大小。
- static long getNativeHeapSize()：返回的是当前进程 navtive 堆本身总的内存大小。

例如下面的演示代码：

```
Log.i(tag,"NativeHeapSizeTotal:"+(Debug.getNativeHeapSize()>>10));
Log.i(tag,"NativeAllocatedHeapSize:"+(Debug.getNativeHeapAllocatedSize()>>10));
Log.i(tag,"NativeAllocatedFree:"+(Debug.getNativeHeapFreeSize()>>10));
```

5. 使用 dumpsys meminfo 命令

可以在 adb shell 中运行 dumpsys meminfo 命令来得到进程的内存信息，在该命令的后面需要加上进程的名字，以确定是哪个进程。比如命令 adb shell dumpsys meminfo com.teleca.robin.test 会得到 com.teleca.robin.test 进程使用的内存的信息：

```
Applications Memory Usage (kB):
Uptime: 12101826 Realtime: 270857936
** MEMINFO in pid 3407 [com.teleca.robin.test] **
                    native   dalvik    other    total
             size:     3456     3139      N/A     6595
        allocated:     3432     2823      N/A     6255
```

```
           free:      23     316      N/A     339
          (Pss):     724    1101     1070    2895
  (shared dirty):     1584    4540     1668    7792
    (priv dirty):      644     608      688    1940
Objects
          Views:       0       ViewRoots:       0
    AppContexts:        0      Activities:       0
         Assets:       3    AssetManagers:       3
   Local Binders:        5    Proxy Binders:      11
Death Recipients:        0
 OpenSSL Sockets:        0
SQL
           heap:       0      memoryUsed:       0
pageCacheOverflo:        0  largestMemAlloc:       0
Asset Allocations
    zip:/data/app/com.teleca.robin.test-1.apk:/resources.arsc: 1K
```

在上述信息中，size 表示的是总内存大小(kb)；allocated 表示的是已使用了的内存大小(kb)；free 表示的是剩余的内存大小(kb)。

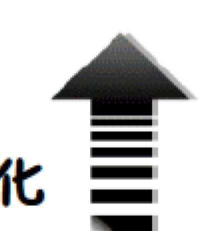
6. 使用 adb shell procrank 命令

如果你想查看所有进程的内存使用情况，可以使用 adb shell procrank 命令。此命令会返回将如下信息：

```
  PID      Vss      Rss      Pss      Uss  cmdline
  188   75832K   51628K   24824K   19028K  system server
  308   50676K   26476K    9839K    6844K  system server
 2834   35896K   31892K    9201K    6740K  com.sec.android.app.twlauncher
  265   28536K   28532K    7985K    5824K  com.android.phone
  100   29052K   29048K    7299K    4984K  zygote
  258   27128K   27124K    7067K    5248K  com.swype.android.inputmethod
  270   25820K   25816K    6752K    5420K  com.android.kineto
 1253   27004K   27000K    6489K    4880K  com.google.android.voicesearch
 2898   26620K   26616K    6204K    3408K
com.google.android.apps.maps:FriendService
  297   26180K   26176K    5886K    4548K  com.google.process.gapps
 3157   24140K   24136K    5191K    4272K  android.process.acore
 2854   23304K   23300K    4067K    2788K  com.android.vending
 3604   22844K   22840K    4036K    3060K  com.wssyncmldm
  592   23372K   23368K    3987K    2812K
com.google.android.googlequicksearchbox
 3000   22768K   22764K    3844K    2724K  com.tmobile.selfhelp
  101    8128K    8124K    3649K    2996K  /system/bin/mediaserver
 3473   21792K   21784K    3103K    2164K  com.android.providers.calendar
 3407   22092K   22088K    2982K    1980K  com.teleca.robin.test
 2840   21380K   21376K    2953K    1996K  com.sec.android.app.controlpanel
```

其实这里的 PSS 和方式四 PSS 的 total 并不一致，这是因为 procrank 命令和 meminfo 命令使用的内核机制不太一样，所以结果会有细微差别。

另外这里的 USS 和方式四的 Priv Dirtyd 的 total 几乎相等，他们似乎表示的是同一个



意义。但是现在得到的关于它们的意义的解释却不太相同。

7. 使用 adb shell cat/proc/meminfo 命令

该方式只能得出系统整个内存的大概使用情况，例如下面的格式：

```
MemTotal:        395144 kB
MemFree:         184936 kB
Buffers:            880 kB
Cached:           84104 kB
SwapCached:           0 kB
```

上述格式中的具体说明如下。

- MemTotal：可供系统和用户使用的总内存大小，它比实际的物理内存要小，因为还有些内存要用于 radio、DMA buffers 等。
- MemFree：剩余的可用内存大小。这里该值比较大，实际上一般 Android system 的该值通常都很小，因为我们尽量让进程都保持运行，这样会耗掉大量内存。
- Cached：是系统用于文件缓冲等的内存，通常 Systems 需要 20MB 大小以避免寻呼状态不好。当内存紧张时，Android 的 Memory Killer 会杀死一些后台进程，以避免他们消耗过多的 cached RAM。

6.3 Android 的内存泄漏

虽然 Dalvik 虚拟机支持垃圾收集，但是这并不意味着可以不用关心内存管理。其实我们更应该格外注意移动设备的内存使用，毕竟其内存空间是受限制的。在实际应用中，一些内存使用问题是很明显的，例如在每次用户触摸屏幕的时候如果应用程序有内存泄漏，将会有可能触发 OutOfMemoryError，最终会导致程序崩溃。另外一些问题却很微妙，也许只是降低应用程序和整个系统的性能(当高频率和长时间地运行垃圾收集器的时候)。本节将详细讲解 Android 系统的内存泄漏问题。

6.3.1 什么是内存泄漏

内存泄漏指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。内存泄漏与许多其他问题有着相似的症状，并且通常情况下只能由那些可以获得程序源代码的程序员才可以分析出来。然而，有不少人习惯于把任何不需要的内存使用的增加描述为内存泄漏，即使严格意义上来说这是不准确的。内存泄漏会因为减少可用内存的数量从而降低计算机的性能。最终，在最糟糕的情况下，过多的可用内存被分配掉导致全部或部分设备停止正常工作，或者应用程序崩溃。

内存泄漏可能不严重，甚至能够被常规的手段检测出来。在现代操作系统中，一个应用程序使用的常规内存在程序终止时被释放。这表示一个短暂运行的应用程序中的内存泄漏不会导致严重后果。



在以下情况，内存泄漏导致较严重的后果：

- 程序运行后置之不理，并且随着时间的流逝消耗越来越多的内存。比如服务器上的后台任务，尤其是嵌入式系统中的后台任务，这些任务可能被运行后很多年内都置之不理；
- 新的内存被频繁地分配，比如当显示电脑游戏或动画视频画面时；
- 程序能够请求未被释放的内存，例如共享内存，甚至是在程序终止的时候；
- 泄漏在操作系统内部发生；
- 泄漏在系统关键驱动中发生；
- 内存非常有限，比如在嵌入式系统或便携设备中；
- 当运行于一个终止时内存并不自动释放的操作系统(比如 AmigaOS)之上，而且一旦丢失只能通过重启来恢复。

6.3.2 为什么会发生内存泄漏

我们知道 JVM 会根据 generation(代)来进行 GC，根据图 6-10 所示，一共被分为 young generation(年轻代)、tenured generation(老年代)、permanent generation(永久代，perm gen)，perm gen(或称 Non-Heap 非堆)是个异类。注意，heap 空间不包括 perm gen。

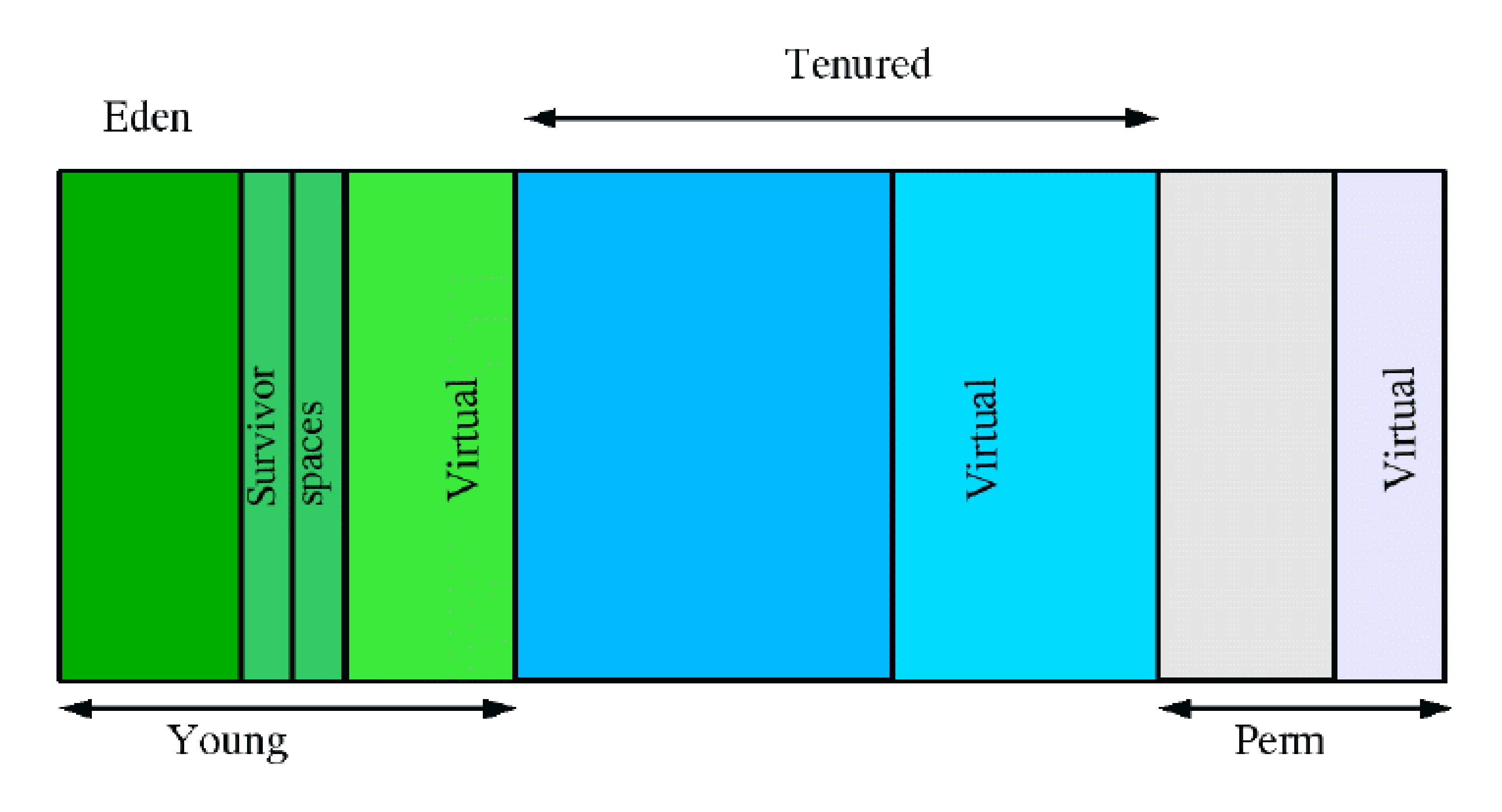


图 6-10 JVM 根据 generation(代)来进行 GC

绝大多数的对象都在 young generation 被分配，也在 young generation 被收回。当 young generation 的空间被填满时，GC 会进行 minor collection(次回收)，这样子的次回收不涉及 heap 中的其他 generation。minor collection 会根据 weak generational hypothesis(弱年代假设)来假设 young generation 中大量的对象都是垃圾需要回收，minor collection 的过程会非常快。在 young generation 中，没有被回收的对象被转移到 tenured generation，然而 tenured generation 也会被填满，最终触发 major collection(主回收)，这次回收针对整个 heap，由于涉及大量对象，所以比 minor collection 慢得多。

JVM 有如下三种垃圾回收器：

- throughput collector：用来做并行 young generation 回收，由参数-XX:+UseParallelGC 启动；
- concurrent low pause collector：用来做 tenured generation 并发回收，由参数 -XX:+UseConcMarkSweepGC 启动；

 Android 内存优化



- incremental low pause collector：是默认的垃圾回收器。不建议直接使用某种垃圾回收器，最好让 JVM 自己决断，除非自己有足够的把握。

Heap 中各 generation 空间是如何划分的呢？通过 JVM 的-Xmx=n 参数可指定最大 heap 空间，而-Xms=n 则是指定最小 heap 空间。在 JVM 初始化的时候，如果最小 heap 空间小于最大 heap 空间的话，如上图所示 JVM 会把未用到的空间标注为 Virtual。除了这两个参数还有-XX:MinHeapFreeRatio=n 和 -XX:MaxHeapFreeRatio=n 来分别控制最大、最小的剩余空间与活动对象之比例。在 32 位 Solaris SPARC 操作系统下，默认值如表 6-1 所示，在 32 位 windows xp 下，默认值也差不多。

表 6-1　各个参数的默认值

参　数	默 认 值
MinHeapFreeRatio	40
MaxHeapFreeRatio	70
-Xms	3670k
-Xmx	64m

由于 tenured generation 的 major collection 过程较慢，所以如果 tenured generation 空间小于 young generation 的话，会造成频繁的 major collection，会影响效率。Server JVM 默认的 young generation 和 tenured generation 空间比例为 1∶2，也就是说 young generation 的 eden 和 survivor 空间之和是整个 heap(当然不包括 perm gen)的 1/3，该比例可以通过-XX:NewRatio=n 参数来控制，而 Client JVM 默认的-XX:NewRatio 是 8。

young generation 中幸存的对象被转移到 tenured generation，但是 concurrent collector 线程在这里进行 major collection，而在回收任务结束前空间被耗尽了，这时将会发生 Full Collections(Full GC)，整个应用程序都会停止下来直到回收完成。由此可见，Full GC 是高负载生产环境的噩梦。

在此还需要说一说异类 perm gen，它是 JVM 用来存储无法在 Java 语言级描述的对象，这些对象分别是类和方法数据(与 class loader 有关)以及 interned strings(字符串驻留)。一般 32 位 OS 下 perm gen 默认 64m，可通过参数-XX:MaxPermSize=n 指定。

接下来回到我们本小节的问题：为何会内存溢出？要回答这个问题又要引出另外一个话题，既什么样的对象 GC 才会回收？当然是 GC 发现通过任何 reference chain(引用链)无法访问某个对象的时候，该对象即被回收。名词 GC Roots 正是分析这一过程的起点，例如 JVM 自己确保了对象的可到达性(那么 JVM 就是 GC Roots)，所以 GC Roots 就是这样在内存中保持对象可到达性的，一旦不可到达，即被回收。通常 GC Roots 是一个在 current thread(当前线程)的 call stack(调用栈)上的对象(例如方法参数和局部变量)，或者是线程自身或者是 system class loader(系统类加载器)加载的类以及 native code(本地代码)保留的活动对象。所以 GC Roots 是分析对象为何还存活于内存中的利器。

从最强到最弱，不同的引用(可到达性)级别反映了对象的生命周期。

- Strong Ref(强引用)：通常我们编写的代码都是 Strong Ref，与此对应的是强可达性，只有去掉强可达，对象才被回收。
- Soft Ref(软引用)：对应软可达性，只要有足够的内存，就一直保持对象，直到发

现内存吃紧且没有 Strong Ref 时才回收对象。一般可用来实现缓存，通过 java.lang.ref.SoftReference 类实现。

- Weak Ref(弱引用)：比 Soft Ref 更弱，当发现不存在 Strong Ref 时，立刻回收对象而不必等到内存吃紧的时候。通过 java.lang.ref.WeakReference 和 java.util.WeakHashMap 类实现。
- Phantom Ref(虚引用)：根本不会在内存中保持任何对象，你只能使用 Phantom Ref 本身。一般用于在进入 finalize()方法后进行特殊的清理过程，通过 java.lang.ref.PhantomReference 实现。

6.3.3 shallow size、retained size

shallow size 是指对象本身占用内存的大小，不包含对其他对象的引用，也就是对象头加成员变量(不是成员变量的值)的总和。在 32 位系统上，对象头占用 8 字节，int 占用 4 字节，不管成员变量(对象或数组)是否引用了其他对象(实例)或者赋值为 null 它始终占用 4 字节。故此，对于 String 对象实例来说，它有三个 int 成员(3*4=12 字节)、一个 char[]成员(1*4=4 字节)以及一个对象头(8 字节)，总共 3*4 +1*4+8=24 字节。根据这一原则，对 String a=”rosen jiang”来说，实例 a 的 shallow size 也是 24 字节。

Retained size 是指该对象自己的 shallow size，加上从该对象能直接或间接访问到对象的 shallow size 之和。换句话说，retained size 是该对象被 GC 之后所能回收到内存的总和。为了更好地理解 retained size，不妨看个例子。

把内存中的对象看成下面图 6-11 中的节点，并且对象和对象之间互相引用。这里有一个特殊的节点 GC Roots，这就是 reference chain 的起点。

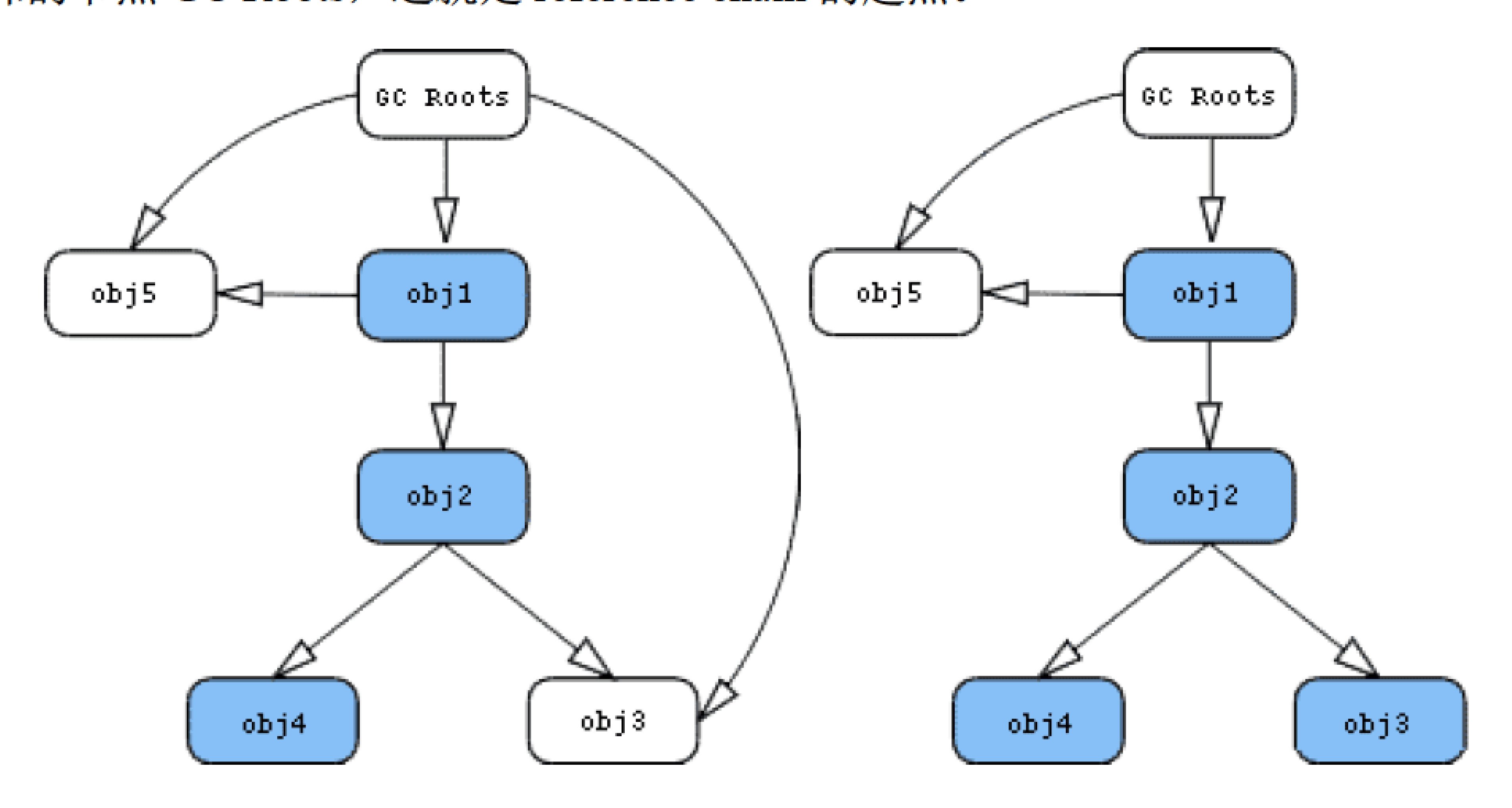


图 6-11 节点图

利用 Strong Ref 存储大量数据，直到 heap 撑破，利用 interned strings(或者 class loader 加载大量的类)把 perm gen 撑破。在图 6-11 中，从 obj1 入手，蓝色节点代表仅仅只有通过 obj1 才能直接或间接访问的对象。因为可以通过 GC Roots 访问，所以左图的 obj3 不是蓝色节点；而在右图却是蓝色，因为它已经被包含在 retained 集合内。所以对于图 6-11 中的

 Android 内存优化



左图来说，obj1 的 retained size 是 obj1、obj2、obj4 的 shallow size 总和；而右图的 retained size 是 obj1、obj2、obj3、obj4 的 shallow size 总和。obj2 的 retained size 可以通过相同的方式计算。

6.3.4 查看 Android 内存泄漏的工具

在开发应用过程中，我们可以使用现成的工具来查看内存泄漏情况。例如 DDMS 和 MAT。有关 DDMS 的知识在本章前面的内容中已经介绍过了，在接下来将讲解 MAT 工具的基本知识。

MAT 是 Memory Analyzer Tool 的缩写，是一个 Eclipse 插件，同时也有单独的 RCP 客户端。笔者使用的是 MAT 的 eclipse 插件，使用插件要比 RCP 稍微方便一些。下载后的目录结构如图 6-12 所示。

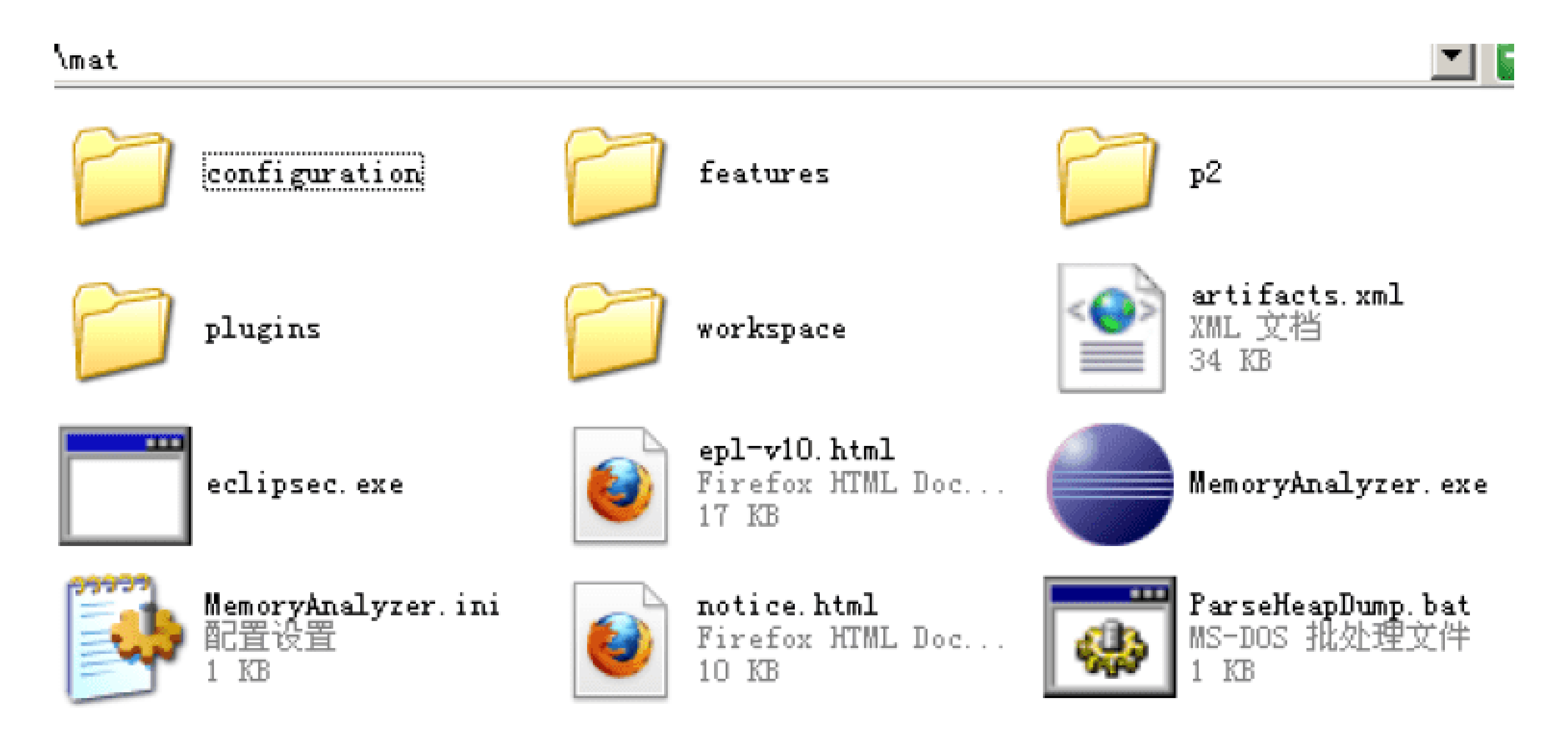


图 6-12　MAT 的文件目录

双击图 6-12 中的 MemoryAnalyzer.exe 可以打开 MAT，打开后的界面如图 6-13 所示。

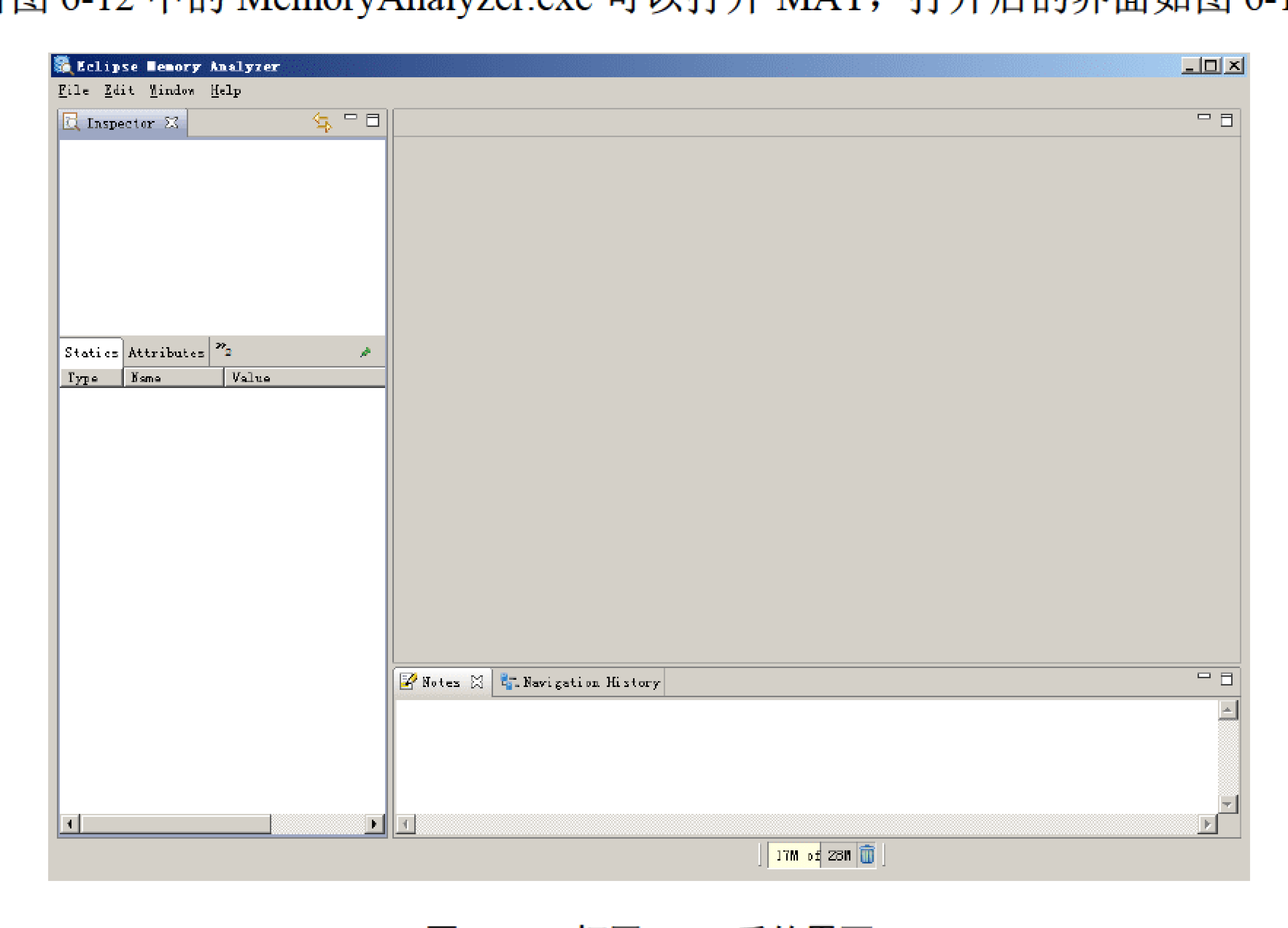


图 6-13　打开 MAT 后的界面

这样通过图 6-13 中的 File 菜单可以打开用 DDMS 生成的.hprof 文件，具体生成.hprof 文件的方法请读者参阅 6.3.5 节中的内容。例如打开一个.hprof 文件后的界面如图 6-14 所示。

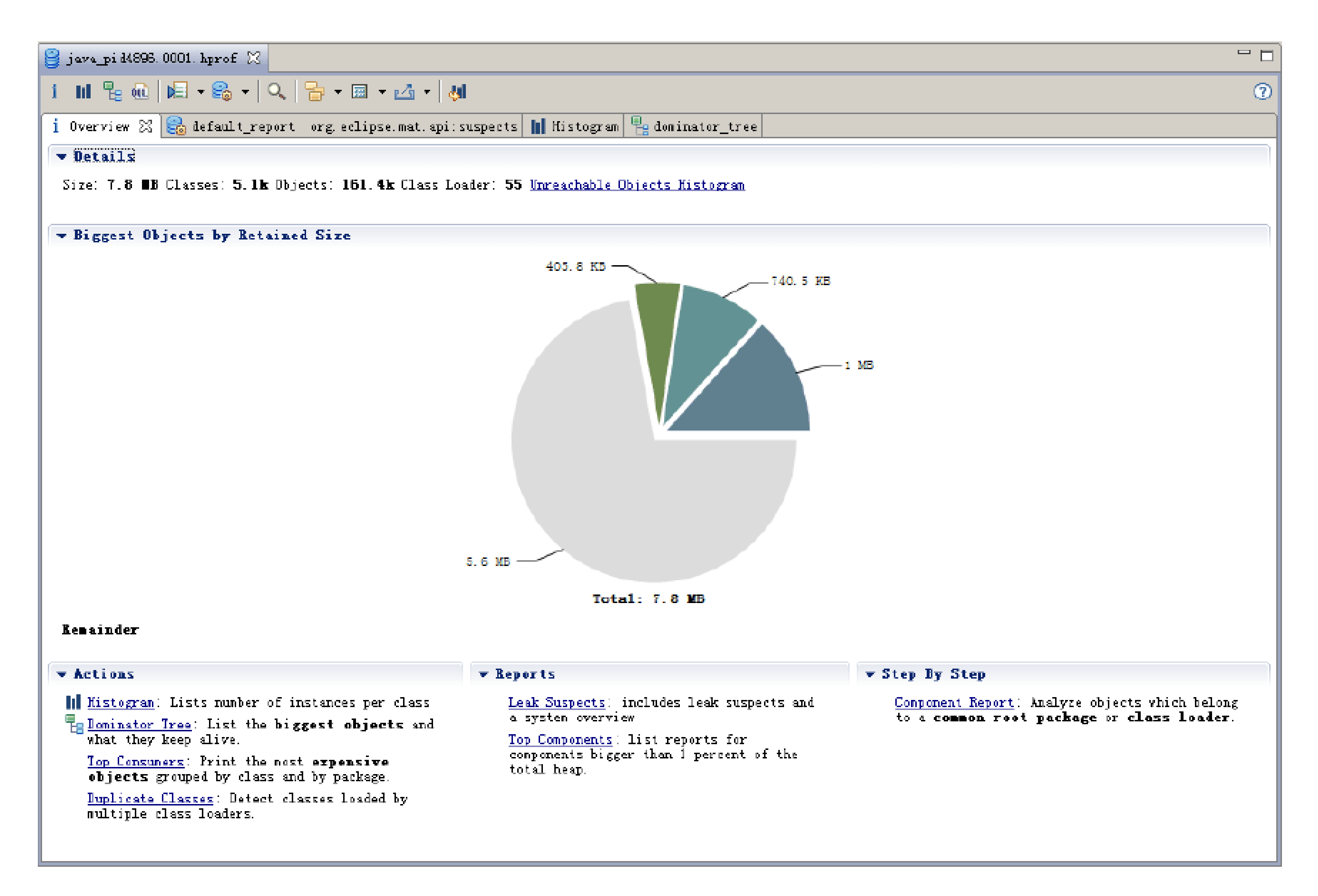


图 6-14 分析界面

从图 6-14 中可以看到 MAT 的大部分功能，具体说明如下。

(1) Histogram：可以列出内存中的对象，对象的个数以及大小。

(2) Dominator Tree 可以列出那个线程，以及线程下面的那些对象占用的空间。

(3) Top consumers 通过图形列出最大的 object。

(4) Leak Suspects 通过 MA 自动分析泄漏的原因。

单击 Histogram 选项后的界面如图 6-15 所示。

java_pid4896.0001.hprof

Overview | default_report org.eclipse.mat.api:suspects | Histogram | dominator_tree

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	33,495	3,035,920	>= 3,035,920
byte[]	1,346	1,293,768	>= 1,293,768
java.lang.String	34,589	830,136	>= 3,395,488
java.util.HashMap$Entry	11,299	271,176	>= 1,348,424
java.util.HashMap$Entry[]	2,657	204,976	>= 1,522,752
java.lang.Object[]	4,655	204,800	>= 1,557,416
java.lang.String[]	3,569	124,656	>= 716,536
int[]	1,938	117,096	>= 117,096
java.util.HashMap	2,607	104,280	>= 1,552,096
org.eclipse.core.internal.registry.ReferenceMap$SoftRef	2,413	96,520	>= 96,520
org.eclipse.core.internal.registry.ConfigurationElement	1,919	92,112	>= 482,248
org.eclipse.osgi.internal.resolver.ExportPackageDescriptionImpl	1,193	76,352	>= 168,752
java.lang.Class	5,085	72,816	>= 2,390,400
java.util.ArrayList	2,711	65,064	>= 223,736
java.util.Hashtable$Entry	2,537	60,888	>= 309,320
org.osgi.framework.Version	1,775	56,800	>= 83,704
java.util.TreeMap$Entry	1,526	48,832	>= 56,888
org.eclipse.osgi.internal.resolver.ImportPackageSpecificationImpl	953	45,744	>= 183,376
java.util.Hashtable$Entry[]	255	32,680	>= 550,680
java.util.LinkedHashMap$Entry	1,007	32,224	>= 107,904
org.eclipse.osgi.internal.module.ResolverImport	953	30,496	>= 60,840
org.eclipse.osgi.service.resolver.StateWire	1,245	29,880	>= 29,880
org.eclipse.osgi.internal.module.ResolverExport	1,185	28,440	>= 47,400
java.lang.Object	3,231	25,848	>= 25,848
org.eclipse.osgi.framework.util.KeyedElement[]	234	25,120	>= 27,176
Total: 25 of 5,076 entries; 5,051 more	161,421	8,136,960	

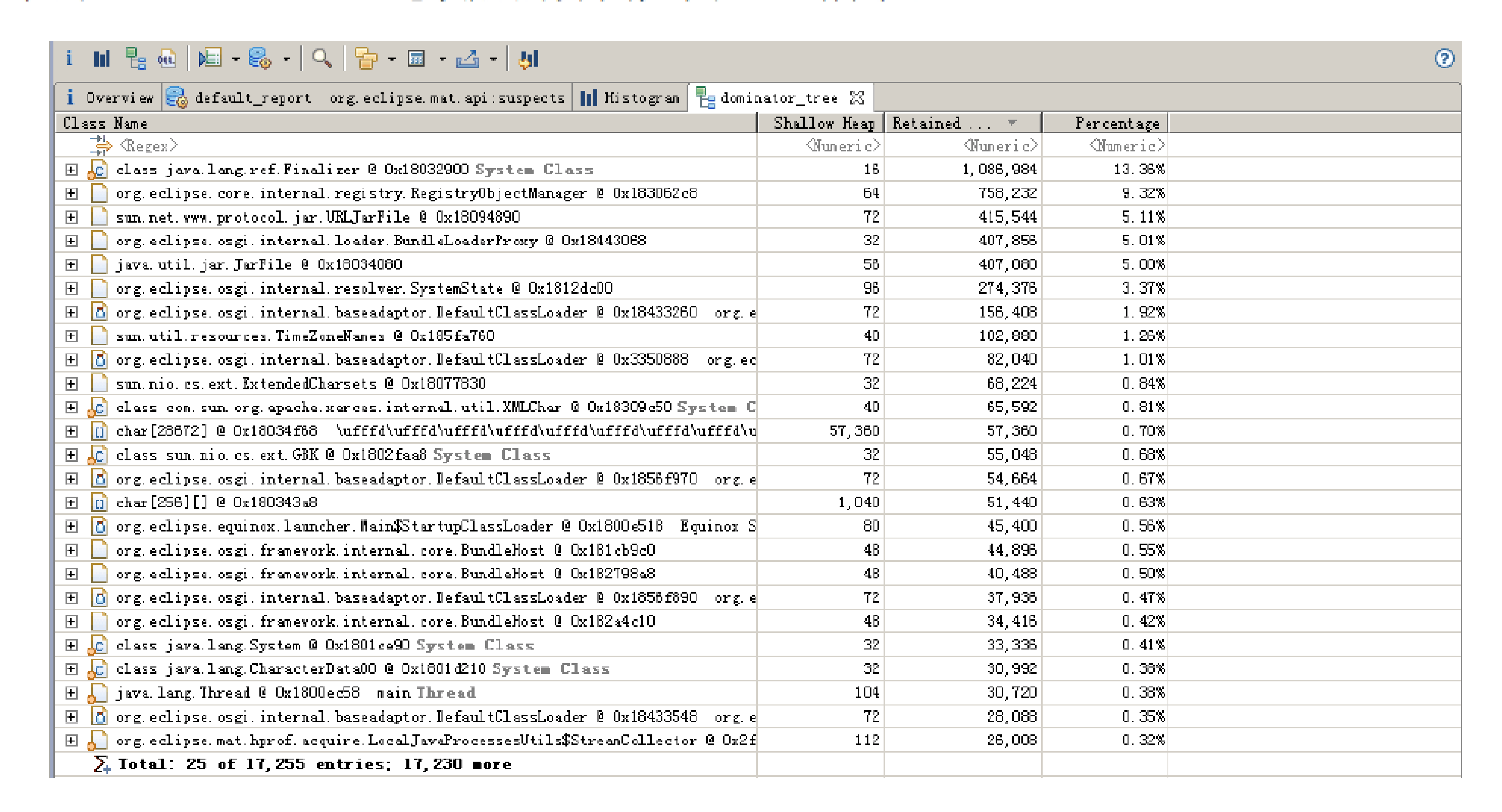
图 6-15 Histogram 界面

 Android 内存优化

图 6-15 中主要选项的说明如下。

- Objects：类的对象的数量。
- Shallow size：就是对象本身占用内存的大小，不包含对其他对象的引用，也就是对象头加成员变量(不是成员变量的值)的总和。
- Retained size：是该对象自己的 shallow size，加上从该对象能直接或间接访问到对象的 shallow size 之和。换句话说，retained size 是该对象被 GC 之后所能回收到内存的总和。

单击 Dominator Tree 选项后的界面如图 6-16 所示。



Overview | default_report org.eclipse.mat.api:suspects | Histogram | dominator_tree

Class Name	Shallow Heap	Retained ...	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
class java.lang.ref.Finalizer @ 0x18032900 System Class	16	1,086,984	13.38%
org.eclipse.core.internal.registry.RegistryObjectManager @ 0x183062c8	64	758,232	9.32%
sun.net.www.protocol.jar.URLJarFile @ 0x18094890	72	415,544	5.11%
org.eclipse.osgi.internal.loader.BundleLoaderProxy @ 0x18443068	32	407,856	5.01%
java.util.jar.JarFile @ 0x18034080	56	407,080	5.00%
org.eclipse.osgi.internal.resolver.SystemState @ 0x1812dc00	96	274,376	3.37%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x18433260 org.e	72	156,408	1.92%
sun.util.resources.TimeZoneNames @ 0x185fa760	40	102,880	1.26%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x3350888 org.ec	72	82,040	1.01%
sun.nio.cs.ext.ExtendedCharsets @ 0x18077830	32	68,224	0.84%
class com.sun.org.apache.xerces.internal.util.XMLChar @ 0x18309c50 System C	40	65,592	0.81%
char[28672] @ 0x18034f68 \ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\u	57,360	57,360	0.70%
class sun.nio.cs.ext.GBK @ 0x1802faa8 System Class	32	55,048	0.68%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x1856f970 org.e	72	54,664	0.67%
char[256][] @ 0x180343a8	1,040	51,440	0.63%
org.eclipse.equinox.launcher.Main$StartupClassLoader @ 0x1800e518 Equinox S	80	45,400	0.56%
org.eclipse.osgi.framework.internal.core.BundleHost @ 0x181cb9c0	48	44,896	0.55%
org.eclipse.osgi.framework.internal.core.BundleHost @ 0x182798a8	48	40,488	0.50%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x1856f890 org.e	72	37,936	0.47%
org.eclipse.osgi.framework.internal.core.BundleHost @ 0x182a4c10	48	34,416	0.42%
class java.lang.System @ 0x1801ce90 System Class	32	33,336	0.41%
class java.lang.CharacterData00 @ 0x1801d210 System Class	32	30,992	0.38%
java.lang.Thread @ 0x1800ec58 main Thread	104	30,720	0.38%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x18433548 org.e	72	28,088	0.35%
org.eclipse.mat.hprof.acquire.LocalJavaProcessesUtils$StreamCollector @ 0x2f	112	26,008	0.32%
Total: 25 of 17,255 entries; 17,230 more			

图 6-16　Dominator Tree 界面

单击 Overview 选项后的界面如图 6-17 所示。

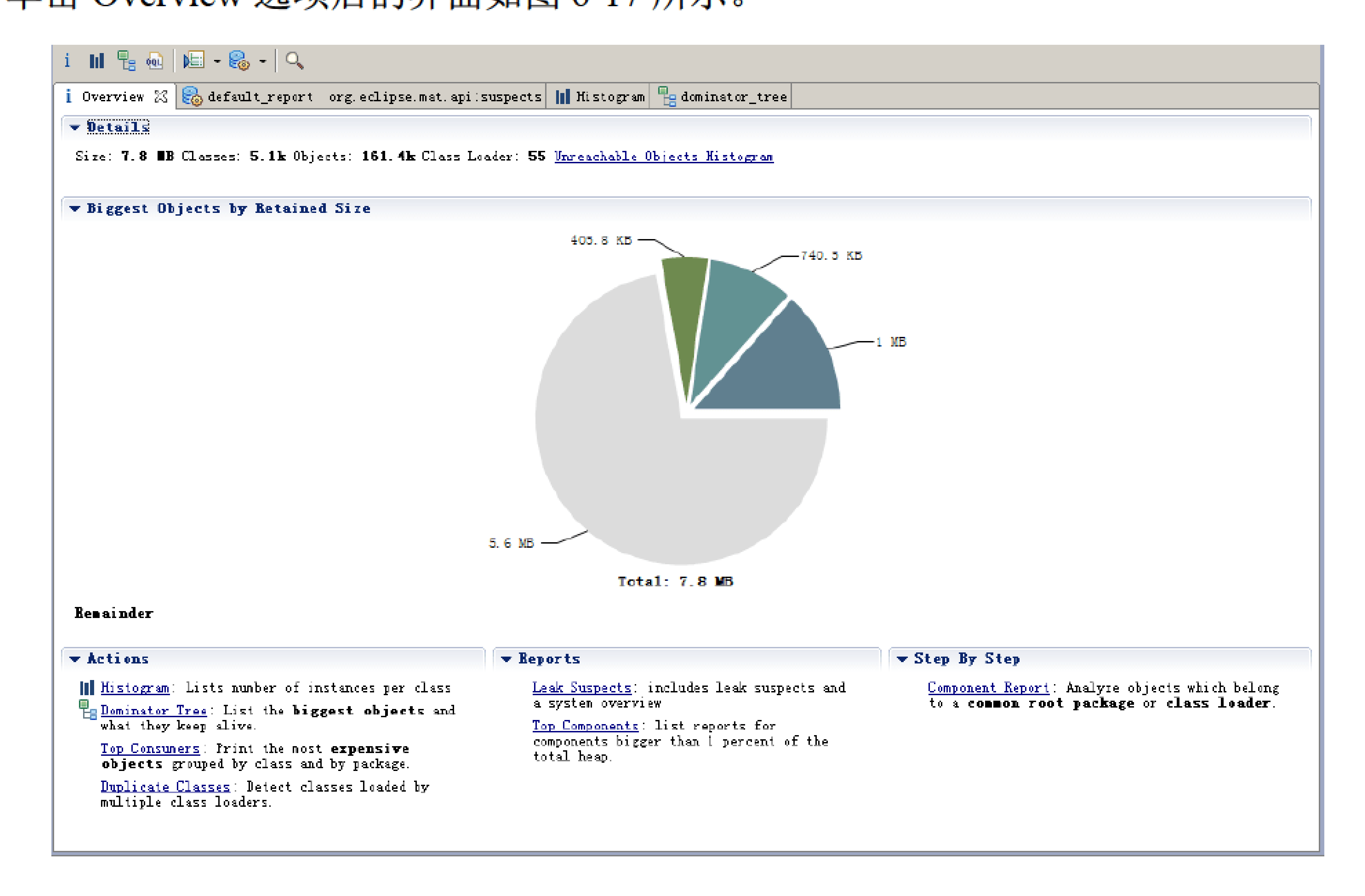


图 6-17　Overview 界面

单击图 6-17 下方的 Leak Suspects 链接后，可以查看详细的内存报表，如图 6-18 所示。

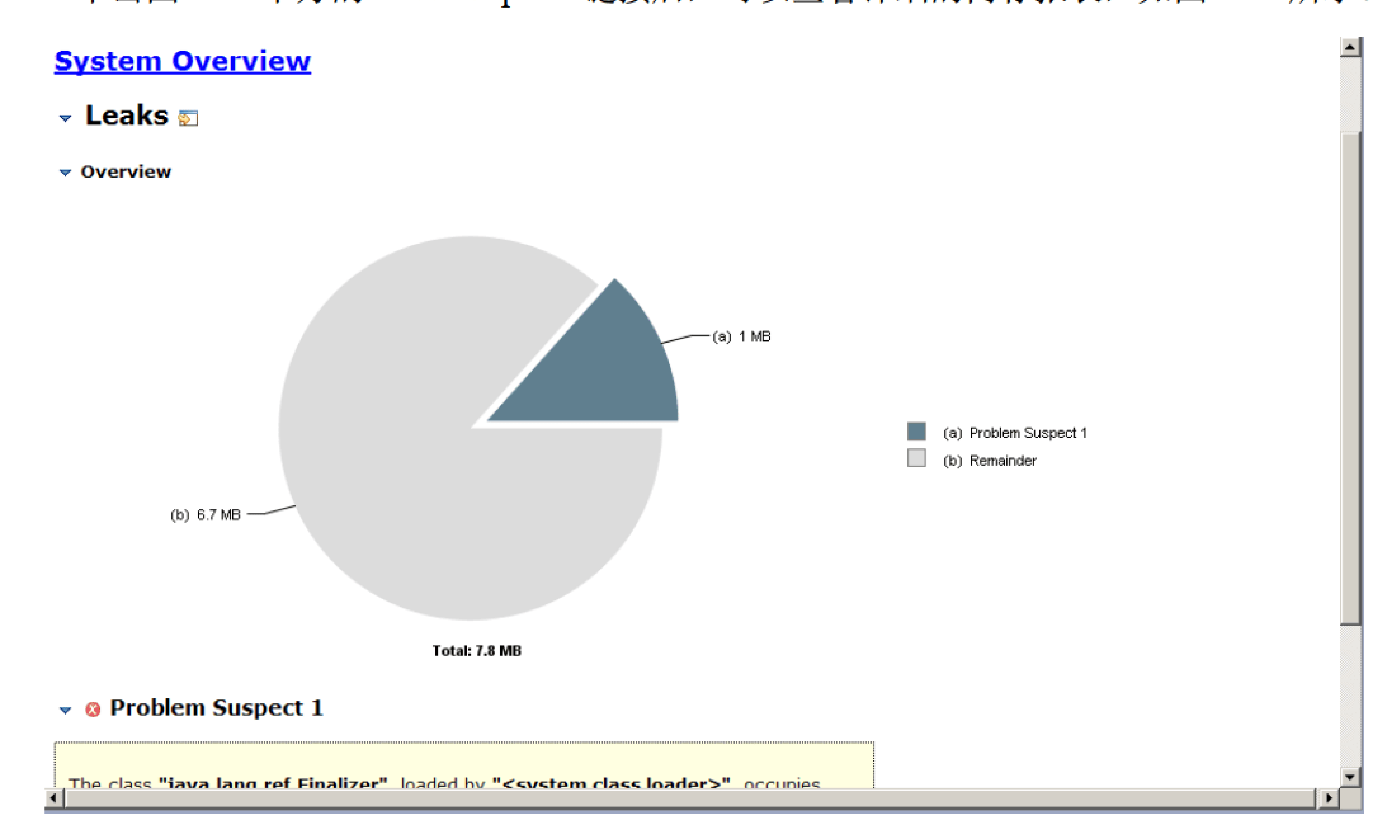


图 6-18　Leak Suspects 查看详细的内存报表

6.3.5　查看 Android 内存泄漏的方法

在日常应用中，通常有如下三种查看 Android 内存泄漏的方法。

1. 生成.hprof 文件

生成.hprof 文件的方法有很多，而且 Android 的不同版本中生成.hprof 的方式也稍有差别，各个版本中生成.prof 文件的方法请参考如下官方网址：

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob plain;f=docs
/heapprofiling.html;hb=HEAD
```

下面以 2.1 版本为例，具体生成流程如下所示。

(1) 打开 Eclipse，切换到 DDMS 透视图，同时确认已经打开了 Devices、Heap 和 logcat 视图；

(2) 将手机设备链接到电脑，并确保使用“USB 调试”模式链接，而不是 Mass Storage 模式；

(3) 当链接成功后，在 Devices 视图中就会看到设备的序列号，和设备中正在运行的部分进程；

(4) 单击选中想要分析的应用的进程，在 Devices 视图上方的一行图标按钮中，同时选中 Update Heap 和 Dump HPROF file 两个按钮；

(5) 这时 DDMS 工具将会自动生成当前选中进程的.hprof 文件，并将其进行转换后存放在 sdcard 当中，如果你已经安装了 MAT 插件，那么此时 MAT 将会自动被启用，并开始对.hprof 文件进行分析；

 Android 内存优化



在上述流程中，第 4 步和第 5 步能够正常使用前提是我们需要有 sdcard，并且当前进程有向 sdcard 中写入的权限(WRITE_EXTERNAL_STORAGE)，否则不会生成.hprof 文件。

在 logcat 中会显示诸如下面的信息：

```
ERROR/dalvikvm(8574): hprof: can't open /sdcard/com.xxx.hprof-hptemp:
Permission denied
```

如果没有 sdcard，或者当前进程没有向 sdcard 写入的权限(如 system_process)，那可以进行如下操作：

(6) 在当前程序中，例如 framework 中某些代码中，可以使用 android.os.Debug 中的如下方法手动的指定.hprof 文件的生成位置。

```
public static void dumpHprofData(String fileName) throws IOException
```

例如：

```
xxxButton.setOnClickListener(new View.OnClickListener() {
 public void onClick(View view)
 {
 android.os.Debug.dumpHprofData("/data/temp/myapp.hprof");
 ... ... }
 }
```

上述代码的功能是，希望在某个按钮被点击的时候开始抓取内存使用信息，并保存在我们指定的位置：/data/temp/myapp.hprof，这样就没有权限的限制了，而且也无须用 sdcard。但是这样做的前提是要保证/data/temp 目录是存在的。这个路径可以自己定义，当然也可以写成 sdcard 当中的某个路径。

2. 使用 MAT 导入.hprof 文件

如果是 Eclipse 自动生成的.hprof 文件，则可以使用 MAT 插件(在后面将讲解这个插件)直接打开(可能是比较新的 ADT 才支持)。如果 Eclipse 自动生成的.hprof 文件不能被 MAT 直接打开，或者是使用 android.os.Debug.dumpHprofData()方法手动生成的.hprof 文件，则需要将.hprof 文件进行转换。为了讲解具体的转换方法，笔者举一个例子，例如将.hprof 文件复制到 PC 上的/ANDROID_SDK/tools 目录下，并输入命令 hprofconv xxx.hprof yyy.hprof，其中 xxx.hprof 表示原始文件，yyy.hprof 为转换过后的文件。转换过后的文件自动放在/ANDROID_SDK/tools 目录下。到此为止，.hprof 文件处理完毕，此时就可以用来分析内存泄漏情况了。

在 Eclipse 中依次单击 Windows | Open Perspective | Other | Memory Analyzer，或者打 Memory Analyzer Tool 的 RCP。在 MAT 中单击 File | Open File，浏览并导入刚刚转换而得到的.hprof 文件。

3. 使用 MAT 的视图工具分析内存

导入.hprof 文件以后，MAT 会自动解析并生成报告，点击 Dominator Tree，并按照 Package 分组，选择自己所定义的 Package 类然后点右键，在弹出菜单中依次选择 List objects | With incoming references。这时会列出所有可疑类，右键点击某一项，并依次选择

Path to GC Roots | exclude weak/soft references，会进一步筛选出跟程序相关的所有有内存泄漏的类。据此，可以追踪到代码中的某一个产生泄漏的类。

具体的分析方法在此不做说明了，因为在 MAT 的官方网站和客户端的帮助文档中有十分详尽的介绍。了解 MAT 中各个视图的作用很重要，例如 www.eclipse.org/mat/about/screenshots.php 中介绍的。

总之使用 MAT 分析内存查找内存泄漏的根本思路，就是找到哪个类的对象的引用没有被释放，找到没有被释放的原因，就可以很容易地定位代码中哪些片段的逻辑有问题了。

另外，在测试过程中首先要分析怎么样操作一个应用会产生内存泄漏，然后在不断的操作中抓取该进程产生的 hhprof 文件，并使用 MAT 工具分析。目前查看内存分析查找内存泄漏的方法还有以下几种。

(1) 使用 top 命令查看某个进程的内存。例如创建一个脚本文件 music.sh，该文件的内容是指定程序每隔一秒钟输出某个进程的内存使用情况，在此具体实现如下：

```
#!/bin/bash
 while true; do
adb shell procrank | grep "com.android.music"
 sleep 1
 done
```

并且配合使用 procank 工具，可以查看 music 进程每一秒钟内存使用的情况。

(2) 另外使用 top 命令也可是查看内存具体为：

```
adb shell top -m 10//查看使用资源最多的 10 个进程
adb shell top|grep com.android.music//查看 music 进程的内存
```

(3) ree 命令。

free 命令用来显示内存的使用情况，使用权限是所有用户。格式如下：

```
free [-b|-k|-m] [-o] [-s delay] [-t] [-V]
```

- 参数－b/－k/－m：表示分别以字节(KB、MB)为单位显示内存使用情况。
- 参数－s delay：显示每隔多少秒数来显示一次内存使用情况。
- 参数－t：显示内存总和列。
- 参数－o：不显示缓冲区调节列。

6.3.6 Android(Java)中常见的容易引起内存泄漏的不良代码

Android 主要应用在嵌入式设备中，而嵌入式设备由于一些众所周知的条件限制，通常都不会有很高的配置，特别是内存是比较有限的。如果我们编写的代码中有太多的对内存使用不当的地方，难免会使得我们的设备运行缓慢，甚至是死机。为了能够使得 Android 应用程序安全且快速地运行，Android 的每个应用程序都会使用一个专有的 Dalvik 虚拟机实例来运行，它是由 Zygote 服务进程孵化出来的，也就是说每个应用程序都是在属于自己的进程中运行的。一方面，如果程序在运行过程中出现了内存泄漏的问题，仅仅会使得自己的进程被 kill 掉，而不会影响其他进程(如果是 system_process 等系统进程出问题的话，则会引起系统重启)。另一方面 Android 为不同类型的进程分配了不同的内存使用上



限，如果应用进程使用的内存超过了这个上限，则会被系统视为内存泄漏，从而被 kill 掉。Android 为应用进程分配的内存上限保存在 ANDROID_SOURCE/system/core/rootdir/init.rc 脚本中，例如下面的部分脚本代码：

```
# Define the oom adj values for the classes of processes that can be
# killed by the kernel.  These are used in ActivityManagerService.
   setprop ro.FOREGROUND_APP_ADJ 0
   setprop ro.VISIBLE_APP_ADJ 1
   setprop ro.SECONDARY_SERVER_ADJ 2
   setprop ro.BACKUP_APP_ADJ 2
   setprop ro.HOME_APP_ADJ 4
   setprop ro.HIDDEN_APP_MIN_ADJ 7
   setprop ro.CONTENT_PROVIDER_ADJ 14
   setprop ro.EMPTY_APP_ADJ 15
# Define the memory thresholds at which the above process classes will
# be killed.  These numbers are in pages (4k).
   setprop ro.FOREGROUND_APP_MEM 1536
   setprop ro.VISIBLE_APP_MEM 2048
   setprop ro.SECONDARY_SERVER_MEM 4096
   setprop ro.BACKUP_APP_MEM 4096
   setprop ro.HOME_APP_MEM 4096
   setprop ro.HIDDEN_APP_MEM 5120
   setprop ro.CONTENT_PROVIDER_MEM 5632
   setprop ro.EMPTY_APP_MEM 6144
# Write value must be consistent with the above properties.
# Note that the driver only supports 6 slots, so we have HOME_APP at the
# same memory level as services.
   write /sys/module/lowmemorykiller/parameters/adj 0,1,2,7,14,15
   write /proc/sys/vm/overcommit_memory 1
   write /proc/sys/vm/min_free_order_shift 4
   write /sys/module/lowmemorykiller/parameters/minfree
1536,2048,4096,5120,5632,6144
   # Set init its forked children's oom_adj.
   write /proc/1/oom_adj -16
```

正因为我们的应用程序能够使用的内存有限，所以在编写代码的时候需要特别注意内存使用问题。如下是一些常见的内存使用不当的情况。

6.4 常见的引起内存泄漏的坏毛病

在本节的内容中，将简单讲解在开发 Android 项目时，因为程序员自身坏毛病而造成内存泄漏的几种情形。希望读者认真学习，为步入本书后面知识的学习打下基础。

6.4.1 查询数据库时忘记关闭游标

程序中经常会进行查询数据库的操作，但是经常会有使用完毕 Cursor 后没有关闭的情

况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在长时间大量操作的情况下才会发现内存问题，这样就会给以后的测试和问题排查带来困难和风险。例如下面的代码：

```
Cursor cursor = getContentResolver().query(uri ...);
if (cursor.moveToNext()) {
    ... ...
}
```

上述代码就是在查询数据库时没有关闭游标，优化修改后的代码如下：

```
Cursor cursor = null;
try {
    cursor = getContentResolver().query(uri ...);
    if (cursor != null && cursor.moveToNext()) {
        ... ...
    }
} finally {
    if (cursor != null) {
        try {
            cursor.close();
        } catch (Exception e) {
          //ignore this
        }
    }
}
```

6.4.2 构造 Adapter 时不习惯使用缓存的 convertView

这也是大多数初学者容易忽视的问题，以构造 ListView 的 BaseAdapter 为例，在 BaseAdapter 中用如下方法向 ListView 提供每一个 item 所需要的 view 对象。

```
public View getView(int position, View convertView, ViewGroup parent)
```

初始时，ListView 会从 BaseAdapter 中根据当前的屏幕布局实例化一定数量的 view 对象，同时 ListView 会将这些 view 对象缓存起来。当向上滚动 ListView 时，原先位于最上面的 list item 的 view 对象会被回收，然后被用来构造新出现的最下面的 list item。这个构造过程就是由 getView()方法完成的，getView()的第二个形参 View convertView 就是被缓存起来的 list item 的 view 对象(初始化时缓存中没有 view 对象则 convertView 是 null)。

由此可以看出，如果不使用 convertView，而是每次都在 getView()中重新实例化一个 View 对象的话，不但浪费资源，而且也浪费时间，也会使得内存占用越来越大。ListView 回收 list item 的 view 对象的过程可以查看 android.widget.AbsListView.java --> void addScrapView(View scrap)方法。

例如下面的演示代码就是不科学的：

```
public View getView(int position, View convertView, ViewGroup parent) {
    View view = new Xxx(...);
    ... ...
```

Android 内存优化

```
    return view;
}
```

优化后的代码如下：

```
public View getView(int position, View convertView, ViewGroup parent) {
    View view = null;
    if (convertView != null) {
        view = convertView;
        populate(view, getItem(position));
        ...
    } else {
        view = new Xxx(...);
        ...
    }
    return view;
}
```

6.4.3 没有及时释放对象的引用

这种情况描述起来比较麻烦，为了说明问题，接下来举两个演示进行说明。

(1) 演示 A

假设有如下操作：

```
public class DemoActivity extends Activity {
    ...
    private Handler mHandler = ...
    private Object obj;
    public void operation() {
     obj = initObj();
     ...
     [Mark]
     mHandler.post(new Runnable() {
            public void run() {
             useObj(obj);
            }
     });
    }
}
```

在上述代码中有一个成员变量 obj，在 operation()中我们希望能够将处理 obj 实例的操作 post 到某个线程的 MessageQueue 中。在上述代码中，即便 mHandler 所在的线程使用完了 obj 所引用的对象，但这个对象仍然不会被垃圾回收掉，因为 DemoActivity.obj 还保有这个对象的引用。所以如果在 DemoActivity 中不再使用这个对象了，可以在[Mark]的位置释放对象的引用，代码可以修改为：

```
public void operation() {
    obj = initObj();
    ...
    final Object o = obj;
```

```
    obj = null;
    mHandler.post(new Runnable() {
        public void run() {
            useObj(o);
        }
    }
}
```

(2) 演示 B

假设我们希望在锁屏界面(LockScreen)中，监听系统中的电话服务以获取一些信息(如信号强度等)，则可以在 LockScreen 中定义一个 PhoneStateListener 的对象，同时将它注册到 TelephonyManager 服务中。对于 LockScreen 对象，当需要显示锁屏界面的时候就会创建一个 LockScreen 对象，而当锁屏界面消失的时候 LockScreen 对象就会被释放掉。

但是如果在释放 LockScreen 对象时，忘记取消之前注册的 PhoneStateListener 对象，则会导致 LockScreen 无法被垃圾回收。如果不断的使锁屏界面显示和消失，则最终会由于大量的 LockScreen 对象没有办法被回收而引起 OutOfMemory，使得 system_process 进程挂掉。

由此可见，当一个生命周期较短的对象 A，被一个生命周期较长的对象 B 保有其引用的情况下，在 A 的生命周期结束时，要及时在 B 中清除掉对 A 的引用。

6.4.4　不在使用 Bitmap 对象时调用 recycle()释放内存

有时我们会手工的操作 Bitmap 对象，如果一个 Bitmap 对象比较占用内存，当它不在被使用的时候，可以调用函数 Bitmap.recycle()回收此对象的像素所占用的内存。但这种做法并不是必需的，需要视具体情况而定。

注意： 除了上述 4 种常见的情形外，Android 应用程序中最典型的需要注意释放资源的情况是在 Activity 的生命周期中，在 onPause()、onStop()、onDestroy()方法中需要适当的释放资源的情况。由于此情况很基础，在此不详细说明，具体可以查看官方文档对 Activity 生命周期的介绍，以明确何时应该释放哪些资源。

6.5　演练解决内存泄漏

Java 编程中经常容易被忽视，但本身又十分重要的一个问题就是内存使用的问题。Android 应用主要使用 Java 语言编写，因此这个问题也同样会在 Android 开发中出现。本节不对 Java 编程问题做探讨，而是对于在 Android 中，特别是应用开发中的此类问题进行整理。

6.5.1　使用 MAT 根据 heap dump 分析 Java 代码内存泄漏的根源

在接下来的内容中，将介绍 MAT 如何根据 heap dump 分析泄漏根源。因为绝大多数



 Android 内存优化

Android 应用程序是用 Java 语言编写的，所以本小节先用一段 Java 代码来测试内存泄漏。这段测试代码非常简单，很容易找出问题，希望读者能够借此举一反三。

一开始不得不说说 ClassLoader，本质上，它的工作就是把磁盘上的类文件读入内存，然后调用 java.lang.ClassLoader.defineClass 方法告诉系统把内存镜像处理成合法的字节码。Java 提供了抽象类 ClassLoader，所有用户自定义类装载器都实例化自 ClassLoader 的子类。system class loader 在没有指定装载器的情况下默认装载用户类，在 Sun Java 1.5 中既 sun.misc.Launcher$AppClassLoader。

(1) 准备 heap dump

请看下面的 Pilot 类的演示代码：

```
package org.rosenjiang.bo;
public class Pilot{
    String name;
    int age;

    public Pilot(String a, int b){
        name = a;
        age = b;
    }
}
```

然后再看类 OOMHeapTest 是如何撑破 heap dump 的。

```
package org.rosenjiang.test;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import org.rosenjiang.bo.Pilot;

public class OOMHeapTest {
    public static void main(String[] args){
        oom();
    }

    private static void oom(){
        Map<String, Pilot> map = new HashMap<String, Pilot>();
        Object[] array = new Object[1000000];
        for(int i=0; i<1000000; i++){
            String d = new Date().toString();
            Pilot p = new Pilot(d, i);
            map.put(i+"rosen jiang", p);
            array[i]=p;
        }
    }
}
```

在上面构造了很多的 Pilot 类实例，然后向数组和 map 中存放。由于是 Strong Ref，GC 自然不会回收这些对象，一直放在 heap 中直到溢出。当然在运行前，先要在 Eclipse 中配置 VM 参数-XX:+HeapDumpOnOutOfMemoryError。好了，一会儿工夫内存溢出，控

制台打出如下信息。

```
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java pid3600.hprof
Heap dump file created [78233961 bytes in 1.995 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

文件 java_pid3600.hprof 就是我们需要的 heap dump，读者可以在 OOMHeapTest 类所在的工程根目录下找到。

(2) 使用 MAT

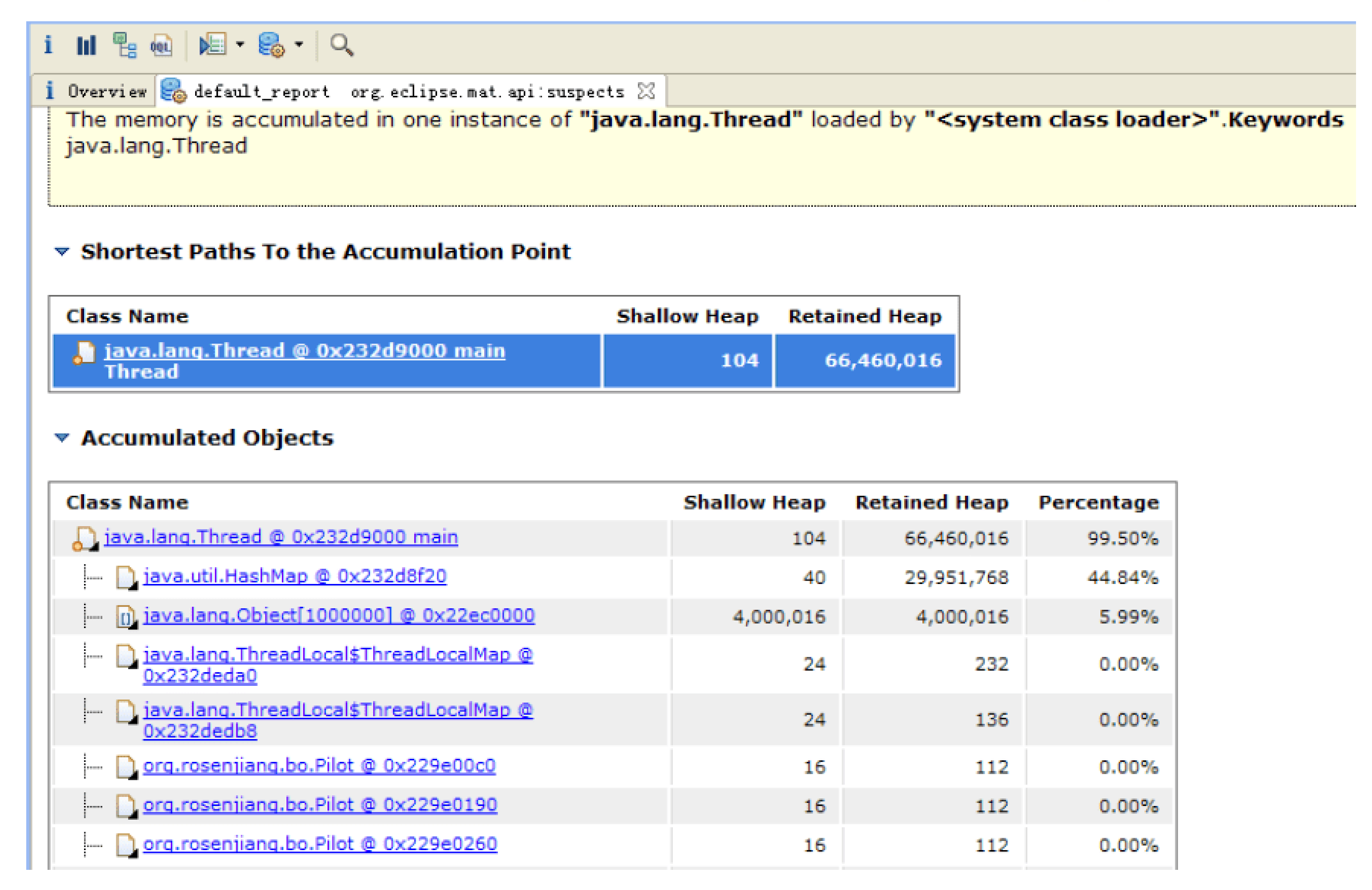
使用 MAT 解析 hprof 文件，弹出向导后直接 Finish 按钮后会看到如图 6-19 所示的界面。



图 6-19 MAT 解析界面

由此可见，通过使用 MAT 工具分析了 heap dump 后，会在界面上非常直观地展示了一个饼图，该图深色区域被怀疑有内存泄漏。我们可以发现整个 heap 才 64M 内存，深色区域就占了 99.5%。接下来是一个简短的描述，告诉我们 main()线程占用了大量内存，并且明确指出 system class loader 加载的 java.lang.Thread 实例有内存聚集，并建议用关键字 java.lang.Thread 进行检查。所以，MAT 通过简单的两句话就说明了问题所在，就算使用者没什么处理内存问题的经验。在下面还有一个 Details 链接，在点开之前不妨考虑一个问题：为何对象实例会聚集在内存中，为何存活(而未被 GC)？是因为 Strong Ref，如图 6-20 所示。

 Android 内存优化



Shortest Paths To the Accumulation Point

Class Name	Shallow Heap	Retained Heap
java.lang.Thread @ 0x232d9000 main Thread	104	66,460,016

Accumulated Objects

Class Name	Shallow Heap	Retained Heap	Percentage
java.lang.Thread @ 0x232d9000 main	104	66,460,016	99.50%
java.util.HashMap @ 0x232d8f20	40	29,951,768	44.84%
java.lang.Object[1000000] @ 0x22ec0000	4,000,016	4,000,016	5.99%
java.lang.ThreadLocal$ThreadLocalMap @ 0x232deda0	24	232	0.00%
java.lang.ThreadLocal$ThreadLocalMap @ 0x232dedb8	24	136	0.00%
org.rosenjiang.bo.Pilot @ 0x229e00c0	16	112	0.00%
org.rosenjiang.bo.Pilot @ 0x229e0190	16	112	0.00%
org.rosenjiang.bo.Pilot @ 0x229e0260	16	112	0.00%

图 6-20　Details 界面

由此可见，单击了 Details 链接之后，除了在上一页看到的描述外，还有 Shortest Paths To the Accumulation Point 和 Accumulated Objects 部分，这里说明了从 GC root 到聚集点的最短路径，以及完整的 reference chain。观察 Accumulated Objects 部分，java.util.HashMap 和 java.lang.Object[1000000]实例的 retained heap(size)最大，我们知道 retained heap 代表从该类实例沿着 reference chain 往下所能收集到的其他类实例的 shallow heap(size)总和，所以明显类实例都聚集在 HashMap 和 Object 数组中了。这里我们发现一个有趣的现象，既 Object 数组的 shallow heap 和 retained heap 一样，数组的 shallow heap 和一般对象(非数组)不同，依赖于数组的长度和里面的元素的类型，对数组求 shallow heap，也就是求数组集合内所有对象的 shallow heap 之和。接下来再来看 org.rosenjiang.bo.Pilot 对象实例的 shallow heap 为何是 16，因为对象头是 8 字节，成员变量 int 是 4 字节，String 引用是 4 字节，所以总共 16 字节。

接下来再来看 Accumulated Objects by Class 区域，如图 6-21 所示。

Accumulated Objects by Class

Label	Number Of Objects	Used Heap Size	Retained Heap Size
org.rosenjiang.bo.Pilot	290,235	4,643,760	32,506,320
java.util.HashMap	1	40	29,951,768
java.lang.Object[]	1	4,000,016	4,000,016
java.lang.String[]	38	1,200	1,200
java.lang.ThreadLocal$ThreadLocalMap	2	48	368
sun.util.calendar.Gregorian$Date	1	96	96
java.lang.StringBuilder	1	16	88
java.security.AccessControlContext	1	24	24
char[]	1	24	24
java.lang.Object	1	8	8
java.lang.Class	1	0	0
∑ Total: 11 entries	290,283	8,645,232	66,459,912

图 6-21　Accumulated Objects by Class 区域

顾名思义，在 Accumulated Objects by Class 区域能找到被聚集的对象实例的类名。此处的类 org.rosenjiang.bo.Pilot 是头条，被实例化了 290 325 次，再返回去看程序，其实是笔者故意而为之。还有很多有用的报告可用来协助分析问题，只是本文中的例子太简单，所以也用不上。

(3) perm gen

perm gen 是一个异类，在里面存储了类和方法数据(与 class loader 有关)以及 interned strings(字符串驻留)。在 heap dump 中没有包含太多的 perm gen 信息。那么我们就用这些少量的信息来解决问题吧。请读者看下面的代码，利用 interned strings 把 perm gen 撑破了。

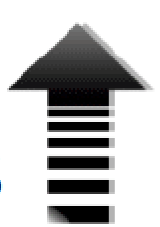
```
package org.rosenjiang.test;
public class OOMPermTest {
    public static void main(String[] args){
        oom();
    }
    private static void oom(){
        Object[] array = new Object[10000000];
        for(int i=0; i<10000000; i++){
            String d = String.valueOf(i).intern();
            array[i]=d;
        }
    }
}
```

控制台会打印如下的信息，然后把 java_pid1824.hprof 文件导入到 MAT。其实在 MAT 里，看到的状况应该和“OutOfMemoryError: Java heap space”差不多(用了数组)，因为 heap dump 并没有包含 interned strings 方面的任何信息。只是在这里需要强调，使用 intern()方法的时候应该多加注意。

```
java.lang.OutOfMemoryError: PermGen space
Dumping heap to java pid1824.hprof
Heap dump file created [121273334 bytes in 2.845 secs]
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
```

开始思考如何把 class loader 撑破，经过尝试会发现使用 ASM 来动态生成类才能达到目的。ASM(http://asm.objectweb.org)的主要作用是处理已编译类(compiled class)，能对已编译类进行生成、转换、分析(功能之一是实现动态代理)，而且它运行起来足够的快和小巧，文档也全面，实属居家必备之良品。ASM 提供了 core API 和 tree API，前者是基于事件的方式，后者是基于对象的方式，类似于 XML 的 SAX、DOM 解析，但是使用 tree API 性能会有损失。到此为止，我们已编译类的结构有：

- 修饰符(例如 public、private)、类名、父类名、接口和 annotation 部分。
- 类成员变量声明，包括每个成员的修饰符、名字、类型和 annotation。
- 方法和构造函数描述，包括修饰符、名字、返回和传入参数类型，以及 annotation。当然还包括这些方法或构造函数的具体 Java 字节码。
- 常量池(constant pool)部分，constant pool 是一个包含类中出现的数字、字符串、类型常量的数组。



已编译类和原来的类源码区别在于：其一，已编译类只包含类本身，内部类不会在已编译类中出现，而是生成另外一个已编译类文件；其二，已编译类中没有注释；其三，已编译类没有 package 和 import 部分。这里还得说说已编译类对 Java 类型的描述：对于原始类型由单个大写字母表示，Z 代表 boolean，C 代表 char，B 代表 byte，S 代表 short，I 代表 int，F 代表 float，J 代表 long，D 代表 double；而对类类型的描述使用内部名(internal name)外加前缀 L 和后面的分号共同表示来表示，所谓内部名就是带全包路径的表示法，例如 String 的内部名是 java/lang/String；对于数组类型，使用单方括号加上数据元素类型的方式描述。最后对于方法的描述，用圆括号来表示，如果返回是 void 用 V 表示，具体参考图 6-22。

Java type	Type descriptor
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Object;
int[]	[I
Object[][]	[[Ljava/lang/Object;

图 6-22 Java 类型的描述

而在下面的代码中会使用 ASM core API，在此需要注意接口 ClassVisitor 是核心，FieldVisitor、MethodVisitor 都是辅助接口。ClassVisitor 应该按照这样的方式来调用：visit visitSource? visitOuterClass? (visitAnnotation | visitAttribute)*(visitInnerClass | visitField | visitMethod)* visitEnd。就是说方法 visit 必须首先调用，再调用最多一次的 visitSource，再调用最多一次的 visitOuterClass 方法，接下来再多次调用 visitAnnotation 和 visitAttribute 方法，最后是多次调用 visitInnerClass、visitField 和 visitMethod 方法。调用完后再调用 visitEnd 方法作为结尾。

另外还需要注意 ClassWriter 类，该类实现了 ClassVisitor 接口，通过 toByteArray 方法可以把已编译类直接构建成二进制形式。由于我们要动态生成子类，所以这里只对 ClassWriter 感兴趣。首先是抽象类原型：

```
package org.rosenjiang.test;
public abstract class MyAbsClass {
    int LESS = -1;
    int EQUAL = 0;
    int GREATER = 1;
    abstract int absTo(Object o);
}
```

其次是自定义类加载器，因为 ClassLoader 的 defineClass 方法都是 protected 的，所以要想加载字节数组形式(因为 toByteArray 了)的类，只有通过继承自己后再实现。

```
package org.rosenjiang.test;

public class MyClassLoader extends ClassLoader {
```

```
    public Class defineClass(String name, byte[] b) {
        return defineClass(name, b, 0, b.length);
    }
}
```

最后看测试类的演示代码：

```
package org.rosenjiang.test;
import java.util.ArrayList;
import java.util.List;
import org.objectweb.asm.ClassWriter;
import org.objectweb.asm.Opcodes;

public class OOMPermTest {
    public static void main(String[] args) {
        OOMPermTest o = new OOMPermTest();
        o.oom();
    }

    private void oom() {
        try {
            ClassWriter cw = new ClassWriter(0);
            cw.visit(Opcodes.V1 5, Opcodes.ACC PUBLIC +
Opcodes.ACC ABSTRACT,
            "org/rosenjiang/test/MyAbsClass", null, "java/lang/Object",
            new String[] {});
            cw.visitField(Opcodes.ACC PUBLIC + Opcodes.ACC FINAL +
Opcodes.ACC STATIC, "LESS", "I",
            null, new Integer(-1)).visitEnd();
            cw.visitField(Opcodes.ACC PUBLIC + Opcodes.ACC FINAL +
Opcodes.ACC STATIC, "EQUAL", "I",
            null, new Integer(0)).visitEnd();
            cw.visitField(Opcodes.ACC PUBLIC + Opcodes.ACC FINAL +
Opcodes.ACC STATIC, "GREATER", "I",
            null, new Integer(1)).visitEnd();
            cw.visitMethod(Opcodes.ACC PUBLIC + Opcodes.ACC ABSTRACT, "absTo",
            "(Ljava/lang/Object;)I", null, null).visitEnd();
            cw.visitEnd();
            byte[] b = cw.toByteArray();

            List<ClassLoader> classLoaders = new ArrayList<ClassLoader>();
            while (true) {
                MyClassLoader classLoader = new MyClassLoader();
                classLoader.defineClass("org.rosenjiang.test.MyAbsClass", b);
                classLoaders.add(classLoader);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

 Android 内存优化

运行后控制台会报错，输出：

```
java.lang.OutOfMemoryError: PermGen space
Dumping heap to java pid3023.hprof
Heap dump file created [92593641 bytes in 2.405 secs]
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
```

打开文件 java_pid3023.hprof，如图 6-23 所示。我们着重看图中的 Classes: 88.1k 和 Class Loader: 87.7k 部分，从这点可看出 class loader 加载了大量的类。

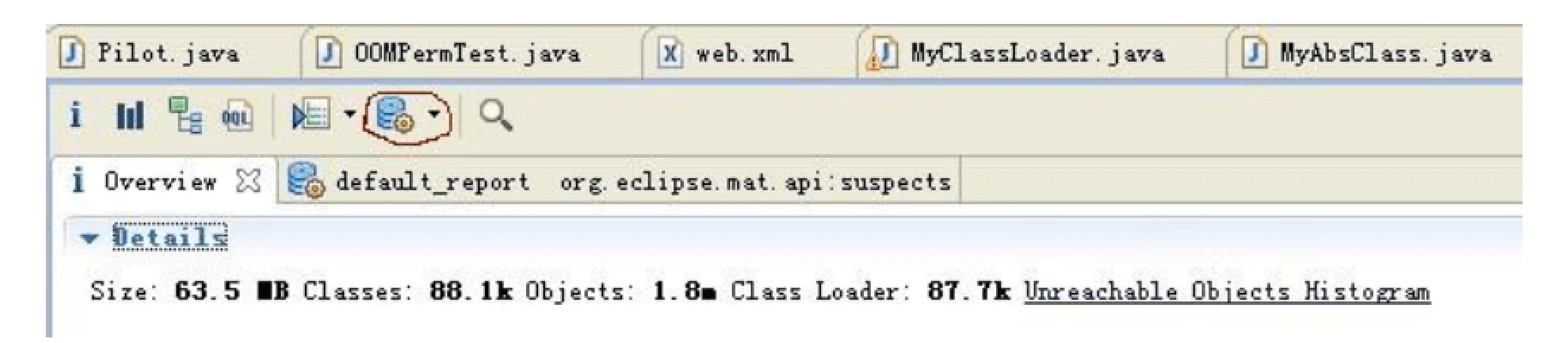


图 6-23　打开文件 java_pid3023.hprof

更进一步分析，需要单击图 6-23 中的按钮，然后选择 Java Basics——Class Loader Explorer 功能。打开后能看到下图所示的界面：第一列是 class loader 名字；第二列是 class loader 已定义类(defined classes)的个数，这里要说一下已定义类和已加载类(loaded classes)了，当需要加载类的时候，相应的 class loader 会首先把请求委派给父 class loader，只有当父 class loader 加载失败后，该 class loader 才会自己定义并加载类，这就是 Java 自己的“双亲委派加载链”结构；第三列是 class loader 所加载的类的实例数目，如图 6-24 所示。

Overview　default_report org.eclipse.mat.api:suspects　classloaderexplorerquery

Class Name	Defined Cla...	No. of Instances
<Regex>	<Numeric>	<Numeric>
<system class loader>	444	1,670,445
sun.misc.Launcher$AppClassLoader @ 0x22efe608	11	87,657
org.rosenjiang.test.MyClassLoader @ 0x229e0148	1	0
parent sun.misc.Launcher$AppClassLoader @ 0x22efe608	11	87,657
org.rosenjiang.test.MyAbsClass		0
Σ Total: 2 entries		
org.rosenjiang.test.MyClassLoader @ 0x229e0428	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e0708	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e09e8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e0cc8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e0fa8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1288	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1568	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1848	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1b28	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1e08	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e20e8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e23c8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e26a8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e2988	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e2c68	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e2f48	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3228	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3508	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e37e8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3ac8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3da8	1	0
Σ Total: 24 of 87,659 entries	**88,110**	**1,758,102**

图 6-24　Class Loader Explorer 功能

在 Class Loader Explorer 面板会发现 class loader 是否加载了过多的类。另外，还有 Duplicate Classes 功能，也能协助分析重复加载的类。我们在此可以肯定的是，MyAbsClass 被重复加载了 N 多次。

注意： 其实 MAT 工具已经非常强大了，我们的上述演示根本用不到 MAT 的其他分析功能。在上述演示中，对于 OOM 不只列举了两种溢出错误，其实还有多种其他错误，但对于 perm gen 来说，如果实在找不出问题所在，建议使用 JVM 的-verbose 参数，该参数会在后台打印出日志，可以用来查看哪个 class loader 加载了什么类，例：[Loaded org.rosenjiang.test.MyAbsClass from org.rosenjiang.test.MyClassLoader]。

6.5.2 演练 Android 中内存泄漏代码优化及检测

在接下来的内容中，将演示测试一个 Android 应用项目内存泄漏的过程。

实例 1	
源码路径	\daima\6\MAT Test
功能	演练 Android 中内存泄漏代码优化及检测

(1) 创建工程

新建 Android 工程 com.devdiv.test.mat_test，新建如下所示的类代码，然后运行该工程。

```
package com.devdiv.test.mat test;

import java.util.ArrayList;
import java.util.List;
import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
   List<String> list = new ArrayList<String>();
//   private PersonInfo person = new PersonInfo();
   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity main);

       new Thread () {
           @Override
           public void run() {
               while (true){
                 MainActivity.this.list.add("OutOfMemoryError soon");
               }
           }
       }.start();
   }
}
```

 Android 内存优化

(2) 分析内存

在此需要获得.hprof 内存镜像文件，我们可以在进程运行过程中切换到 DDMS 的透视图页面，然后选中要查看内存镜像的进程，并单击 Dump HPROF file 即可，如图 6-25 所示。

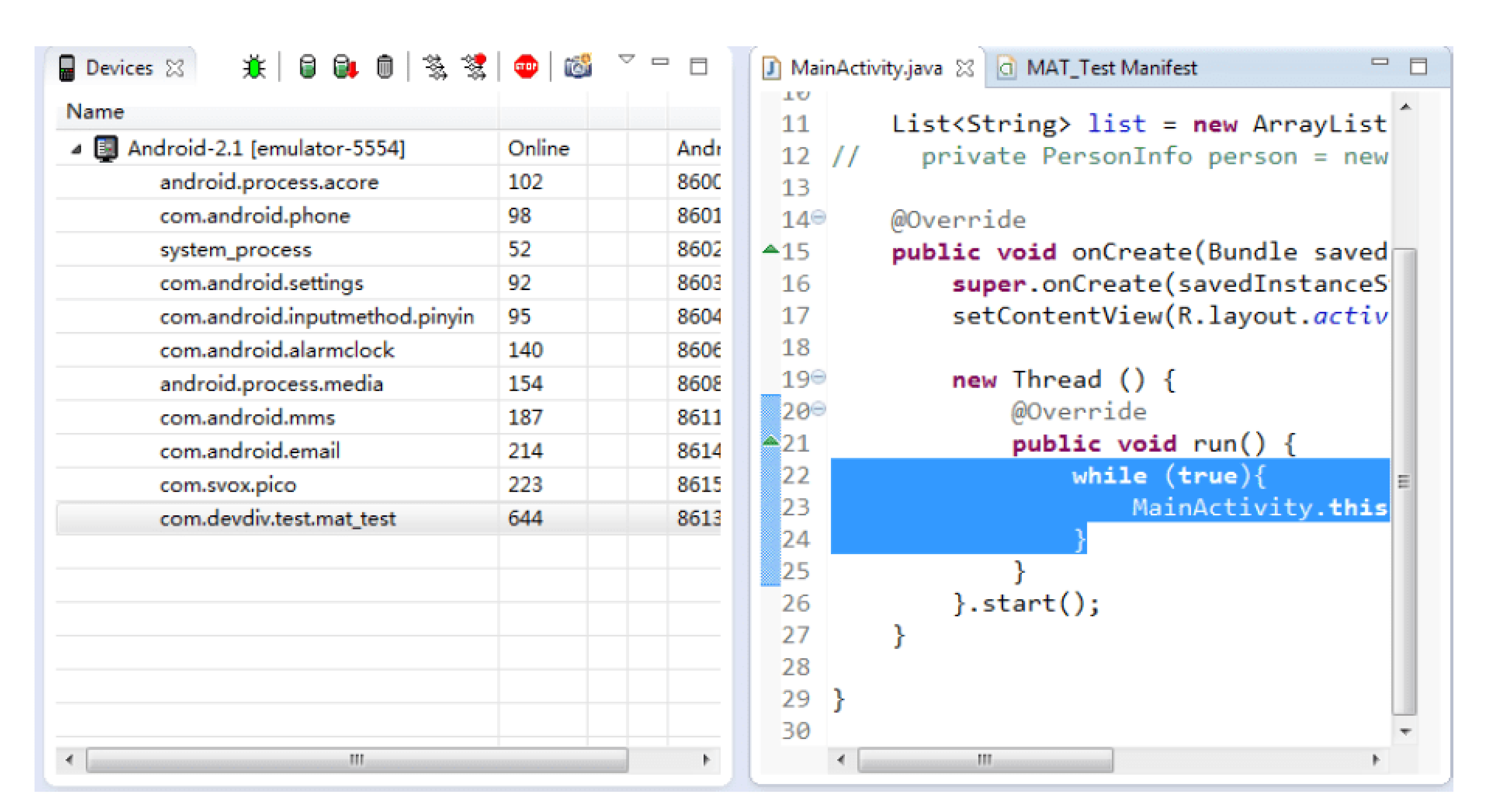


图 6-25　内存镜像文件分析

生成的 hprof 文件会默认使用 MAT 打开，选择 Leak Suspects Report 后，单击 Finish 按钮，如图 6-26 所示。

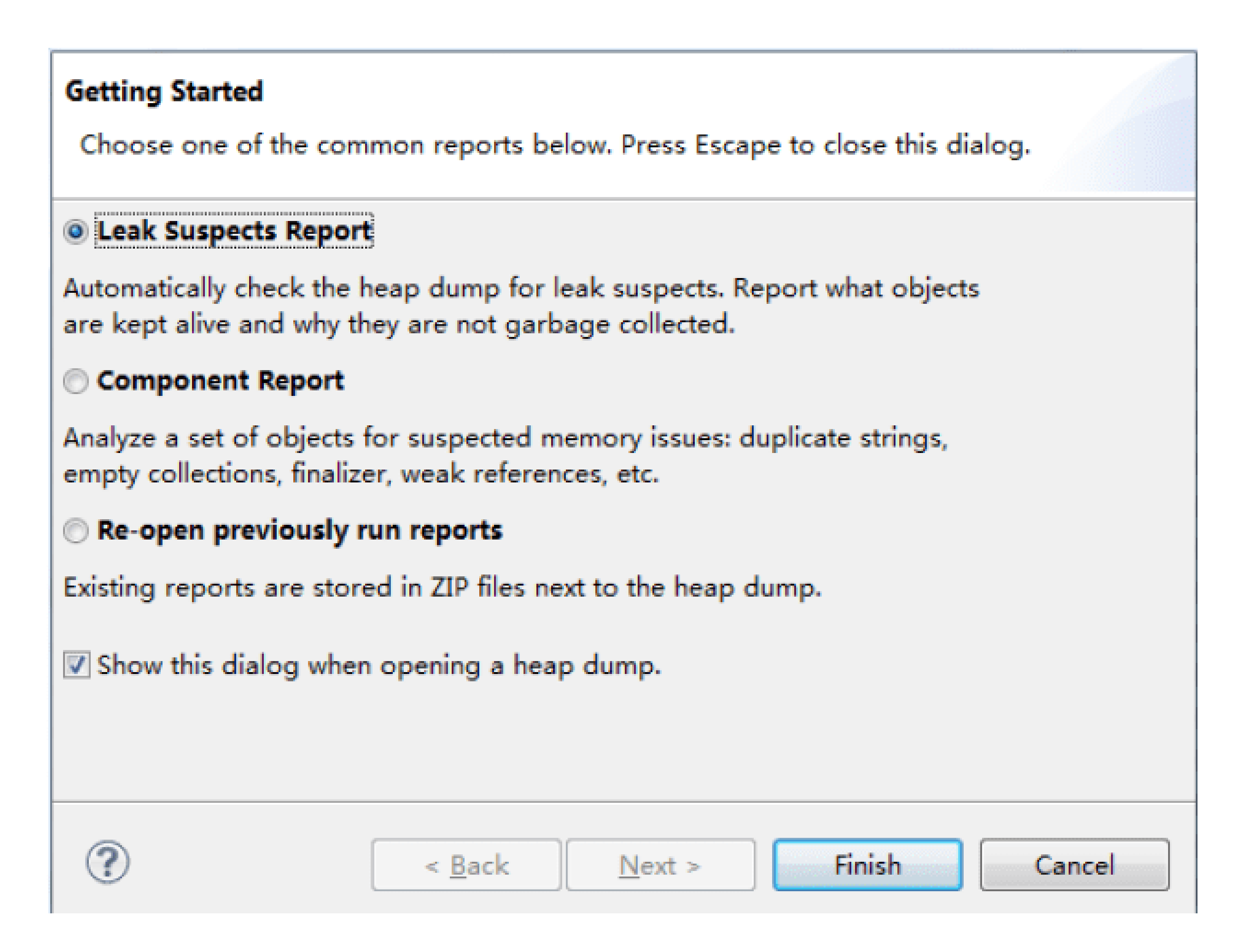


图 6-26　用 MAT 打开内存镜像文件

经过一段时间的初始化后，就能够直观地看到关于内存泄漏的饼图，如图 6-27 所示。

然后就可以查看相关的内存泄漏，如图 6-28 所示。

这样通过提示就可以找到内存泄漏所在，然后就可以根据班长前面内容所讲进行一一优化。

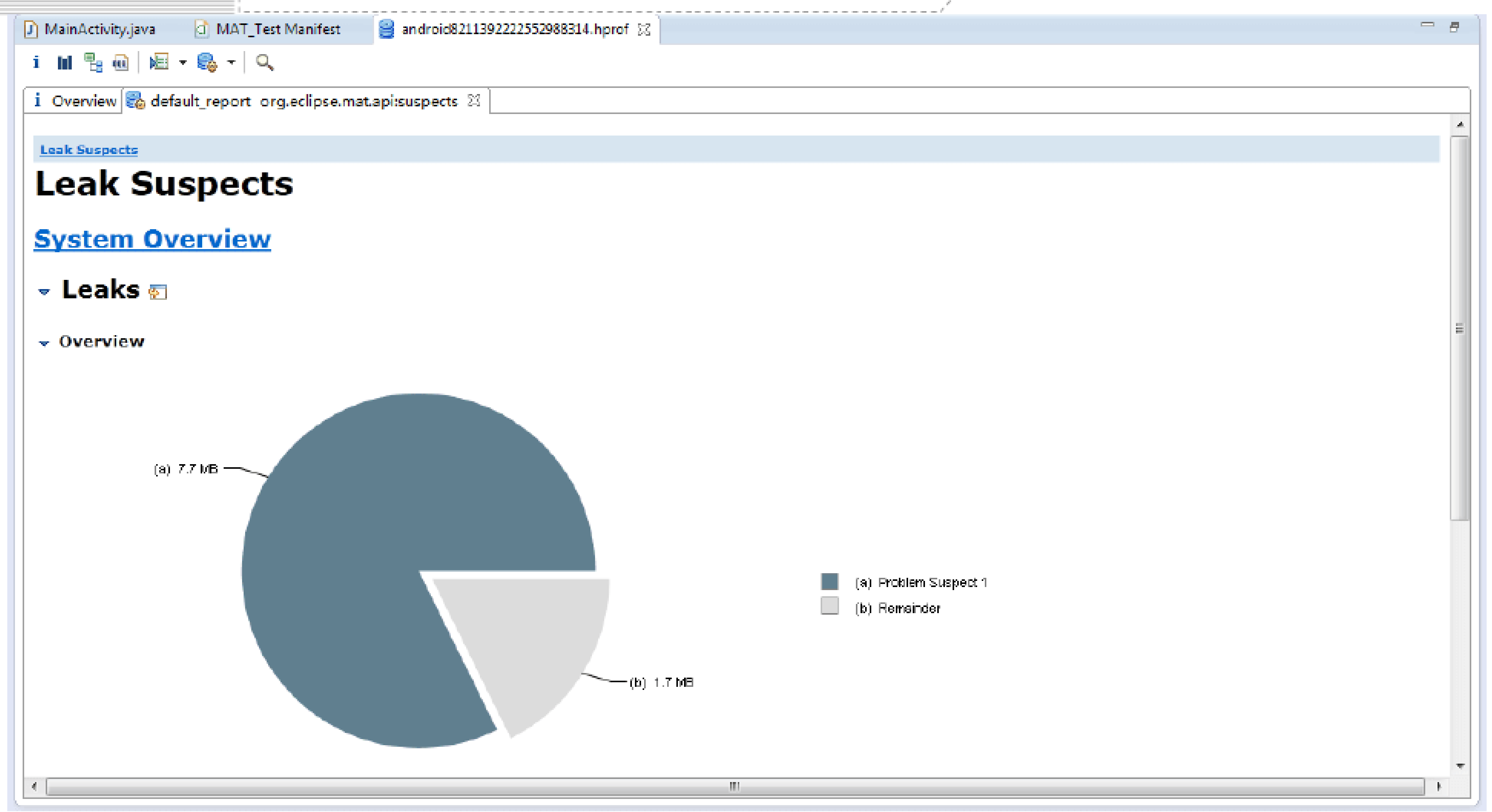
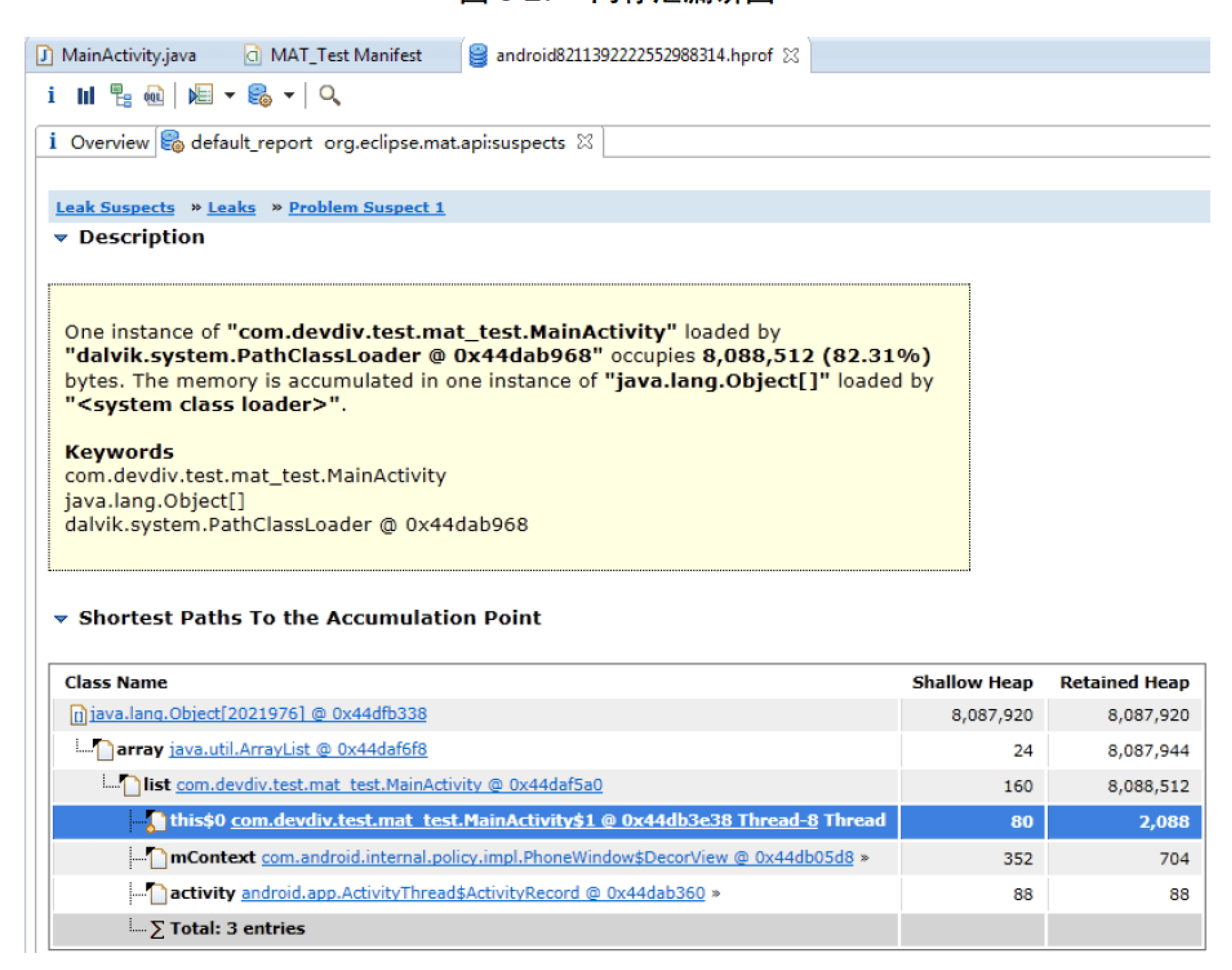


图 6-27　内存泄漏饼图



Class Name	Shallow Heap	Retained Heap
java.lang.Object[2021976] @ 0x44dfb338	8,087,920	8,087,920
array java.util.ArrayList @ 0x44daf6f8	24	8,087,944
list com.devdiv.test.mat_test.MainActivity @ 0x44daf5a0	160	8,088,512
this$0 com.devdiv.test.mat_test.MainActivity$1 @ 0x44db3e38 Thread-8 Thread	80	2,088
mContext com.android.internal.policy.impl.PhoneWindow$DecorView @ 0x44db05d8 »	352	704
activity android.app.ActivityThread$ActivityRecord @ 0x44dab360 »	88	88
∑ Total: 3 entries		

图 6-28　内存泄漏树形图

6.6　Android 图片的内存优化

在 Android 应用中，当对图片本身进行操作时，应该尽量不要使用 setImageBitmap、setImageResource、BitmapFactory.decodeResource 来设置一张大图，因为这些方法在完成

Android 内存优化



Decode 后，最终都是通过 Java 层的 createBitmap()方法来完成的，这需要消耗更多内存。因此，应该先通过 BitmapFactory.decodeStream 方法创建出一个 bitmap，然后再将其设为 ImageView 的 source。decodeStream 最大的优点是直接调用 JNI>>nativeDecodeAsset()来完成 decode，而无需再使用 Java 层的 createBitmap，从而节省了 Java 层的空间。如果在读取时加上图片的 Config 参数，可以更有效的减少加载的内存，从而更有效地阻止抛出内存异常。另外，decodeStream()直接用图片来读取字节码，不会根据机器的各种分辨率来自动适应。当使用了 decodeStream()后，需要在 hdpi 和 mdpi 中配置相应的图片资源，否则在不同分辨率机器上都是同样大小(像素点数量)，显示出来的大小就不对了。

请读者看下面的演示代码：

```
InputStream is = this.getResources().openRawResource(R.drawable.pic1);
 BitmapFactory.Options options=new BitmapFactory.Options();
 options.inJustDecodeBounds = false;
 options.inSampleSize = 10;   //width，hight 设为原来的十分一
 Bitmap btp =BitmapFactory.decodeStream(is,null,options);

if(!bmp.isRecycle() ){
        bmp.recycle()   //回收图片所占的内存
        system.gc()  //提醒系统及时回收
}

/**
  * 以最省内存的方式读取本地资源的图片
  * @param context
  * @param resId
  * @return
  */
publicstatic Bitmap readBitMap(Context context, int resId){
     BitmapFactory.Options opt = new BitmapFactory.Options();
     opt.inPreferredConfig = Bitmap.Config.RGB_565;
     opt.inPurgeable = true;
     opt.inInputShareable = true;
//获取资源图片
     InputStream is = context.getResources().openRawResource(resId);
return BitmapFactory.decodeStream(is,null,opt);
 }
```

在上述代码中，option 中的值指的是对图片进行的缩放比例，SDK 中建议其值是 2 的指数值，如果值越大，越会导致图片不清晰。

我们需要优化 Dalvik 虚拟机的堆内存分配。对于 Android 平台来说，其托管层使用的是 Dalvik Java VM，从目前的表现来看还有很多地方可以优化处理，比如在开发一些大型游戏或耗资源的应用中可能会考虑用手动干涉 GC 处理，使用类 dalvik.system.VMRuntime 提供的方法 setTargetHeapUtilization 可以增强程序堆内存的处理效率。使用如下方法即可：

```
private final static float TARGET_HEAP_UTILIZATION = 0.75f;
VMRuntime.getRuntime().setTargetHeapUtilization(TARGET_HEAP_UTILIZATION);
```

另外还可以用如下方法定义堆内存的大小，这样就实现了优化功能。

```
private final static int CWJ HEAP SIZE = 6* 1024* 1024 ;
VMRuntime.getRuntime().setMinimumHeapSize(CWJ HEAP SIZE);
 //设置最小 heap 内存为 6MB 大小
```

Android

第7章

代码优化

对于程序开发来说，就是一个架构并具体编码的过程，其中编码工作占据了一个项目所需工作的相当比重。在编写Android 代码时，我们需要编写最科学合理的代码，只有这样才能提高程序的效率，达到优化的目的。对于 Android 应用来说，基本上是用 Java 语言来实现的，所以在本章的内容中，有相当一部分的内容是讲解 Java 的代码优化知识。希望读者专心学习本章内容，为步入本书后面高级知识的学习打下基础。

7.1 Android 代码优化的基本原则

在讲解本章的核心内容之前，先向广大读者介绍 Android 代码优化的基本原则。总体原则是：不做不必要的事，不分配不必要的内存。具体来说，主要有如下 15 条原则。

(1) 字符串频繁操作时，多用 StringBuffer 而少用 String。

(2) 尽量使用本地变量，即反复使用的变量要先保存成临时或局部变量，尤其是循环中使用的变量。

(3) String 方法中 substring 和 indexOf 都是 Native(本地)方法，可以大量的使用。

(4) 如果函数返回了 String 类型，而且返回后的使用就是要加入到 StringBuffer，此时可以直接传入 StringBuffer。

(5) 用两个一维数组代替二维数组，例如用 int[] int[]替换 int[][]，因为这两者是等价的。

(6) 如果返回直接类型足够了，就不应返回接口类型，如返回 Hashmap 就足够了，请不要返回 Map。

(7) 如果一个方法不访问(不修改)成员变量，请用 static 方法。

(8) 尽量不用 getters 和 setters，如果你非要用的话请加上 final 关键字，编译器会把它当成内联函数。

(9) 永远不要在 for 循环第二个参数中使用方法调用。

(10) 不修改的 static 变量，请用 static final 常量代替。

(11) foreach 可以用来处理数组和 arraylist，如果处理其他对象相当于 Iterator。

(12) 避免使用枚举，请使用常量代替。

(13) 慎用浮点数 float 尤其是大量的数学运算。

(14) 不使用的引用变量要手动置 null，提高内存被回收的概率。

(15) 慎用图片操作，使用后要立即释放资源。

7.2 优化 Java 代码

因为大多数 Android 应用程序是用 Java 语言编写的，所有用经过优化的 Java 代码编写的 Android 程序，可以提高 Android 程序的执行效率，从而达到提高用户体验的目的。本节将简要介绍优化 Java 代码的基本知识。

7.2.1 GC 对象优化

Java 程序中的内存管理机制是通过 GC 完成的(这一点和 Android 一样)，“一个对象创建后被放置在 JVM 的堆内存中，当永远不再应用这个对象的时候将会被 JVM 在堆内存中回收。被创建的对象不能再生，同时也没有办法通过程序语句释放”，这是《Java 的 GC



机制》中提到的定义。意思是：在运行环境中，JVM 会对两种内存进行管理，一种是堆内存(对象实例或者变量)，一种是栈内存(静态或非静态方法)。而 JVM 所管理的内存区域实际上就是堆内存＋栈内存(对象实例＋实例化变量＋静态方法＋非静态方法)。当 JVM 在其所管理的内存区域中无法通过根集合到达对象的时候就会将此对象作为垃圾对象实施回收。

根据上述定义，可以总结出如下对 Java 的优化原则。

(1) 循环优化

例如下面的代码会一直去执行 alist.size()方法，带来性能消耗。

```
List alist=uSvr.getUserinfoList();
        for(int i=0;i<alist.size();i++){
}
```

应该修改为：

```
for(int i=0 p=alist.size();i<p;i++){
}
```

(2) 循环内不要创建对象

例如在下面的代码中，会在内存中保存 N 份这个对象的引用，这样会浪费大量的内存空间。

```
AuditResult auditResult;
for(int i=1;i<=domainCount;i++){
auditResult=new AuditResult();
……
}
```

应该修改为：

```
AuditResult auditResult;
for(int i=1;i<=domainCount;i++){
auditResult=new AuditResult();
……
}
```

(3) “消灭”不可视阶段的对象

究竟可以将什么样的对象认定为不可视阶段呢？举一个例子吧，在如下代码段中：

```
try{...
}
catch(Exception){
...}
```

如果在 try 的代码块中声明了一个 obj，那么当整个上述代码段执行完毕以后，其实这个 obj 实际上就已经属于不可视阶段了。此时我们应该修改为：

```
try{
  Object obj=new Object();
}catch(Excepione e){
obj=null;
}
```

(4) 少用 new 创建对象

当使用关键字 new 创建类的实例时，构造函数链中的所有构造函数都会被自动调用。但如果一个对象实现了 Cloneable 接口，我们可以调用它的 clone()方法。clone()方法不会调用任何类构造函数。当在使用设计模式(Design Pattern)的场合，如果用 Factory 模式创建对象，则应该用方法 clone()创建新的对象实例。例如，下面是 Factory 模式的一个典型实现：

```
public static Credit getNewCredit()
{
    return new Credit();
}
```

应该改为：

```
private static Credit BaseCredit = new Credit();
  public static Credit getNewCredit()  {
   return (Credit) BaseCredit.clone();
  }
```

这样当 new 创建对象不可避免时，需要注意避免多次使用 new 初始化一个对象。而是应该尽量在使用时再创建该对象。例如下面的演示代码：

```
NewObject object = new NewObject();
int value;
if(i>0 )
{
 value =object.getValue();
}
```

应该修改为：

```
int value;
if(i>0 )
{
 NewObject object = new NewObject();
 Value =object.getValue();
}
```

(5) 及时清除 Session

在通常情况下，当达到设定的超时时间时，同时有些 Session 没有了活动，服务器会释放这些没有活动的 Session。不过在这种情况下，特别是多用户并访时，系统内存要维护多个无效的 Session。当用户退出时，应该立即手动释放，并回收资源。例如和下面的演示代码那样。

```
HttpSession theSession = request.getSession();
// 获取当前 Session
if(theSession != null){
theSession.invalidate(); // 使该 Session 失效
}
```

(6) 乘法和除法问题

请读者看下面的代码：

```
for (val = 0; val &lt; 100000; val +=5) {
 shiftX = val 8;
 myRaise = val 2;
}
```

如果我们利用位移(bit)来处理，性能将会六倍增加。重写后的代码如下：

```
for (val = 0; val &lt; 100000; val += 5) {
 shiftX = val &lt;&lt; 3;
 myRaise = val &lt;&lt; 1;
 }
```

这样移位代替了乘以 8，同样我们可以使用同等效果的左移 3 位。每一个移动相当于乘以 2，变量 myRaise 对此做了证明。同样向右移位相当于除以 2，这样会使执行速度加快。

(7) 用代码处理内存溢出

在 Java 程序中，OutOfMemoryError 是由于内存不够而遇到的一个问题。例如下面的一段代码能有效判断内存溢出错误，并在内存溢出发生时有效回收内存。

```
import java.util.*;
public class DataServer{
private Hashtable data = new Hashtable();
public Object get (String key)
{
Object obj = data.get (key);
if (obj == null)
{
System.out.print (key + " ");
try
{
// simulate getting lots of data
obj = new Double[1000000];
data.put (key, obj);
}
catch (OutOfMemoryError e)
{
System.out.print ("/No Memory! ");
flushCache();
obj = get (key);// try again
}
}
return (obj);
}
public void flushCache()
{
System.out.println ("Clearing cache");
data.clear();
}
```

```
public static void main (String[] args)
{
DataServer ds = new DataServer();
int count = 0;
while (true) // infinite loop for test
ds.get ("" count+);
}
}
```

通过上述代码，可以联想到有效管理连接池溢出的原理。

7.2.2 尽量使用 StringBuilder 和 StringBuffer 进行字符串连接

讲一个笔者的亲身经历：有一天在做性能测试的时候，发现了一个 Web 端 CPU 的性能出现骤降的问题，但是一直没有找到原因。最初我怀疑是和 Tomcat 的线程数有关，后来又怀疑跟数据库的响应时间太长有关系，到最后都一一排除了。之所以此问题比较难以定位，主要是因为通过现有的监控工具无法获知和分析 Tomcat 内部各个线程的占用资源的情况。后来我安装了 Jprofiler，然后又重新进行了一次压力测试，终于找到了问题的根源，原来主要的资源消耗点是在字符串的拼接上，在我的代码中使用了“+”来连接字符串。

在 Java 程序中，可以通过 String、StringBuffer 和 SrtingBuilder 三个对象实现连接字符串的功能。接下来我们来测试究竟谁的效率更高。因为很多高手建议：避免使用 String 通过“+”连接字符串，特别是连接的次数很多的时候，一定要用 StringBuffer，但究竟效率多高，速度多快，接下来我们进行具体测试。测试代码如下。

```
public class TestStringConnection {
   //连接时间的设定
   private final static int n = 20000;
   public static void main(String[] args){
      TestStringConnection test = new TestStringConnection ();
      test.testStringTime(n);
      test.testStringBufferTime(n);
      test.testStringBuilderTime(n);
//       //连接 10 次
//       test.testStringTime(10);
//       test.testStringBufferTime(10);
//       test.testStringBuilderTime(10);
//       //连接 100
//       test.testStringTime(100);
//       test.testStringBufferTime(100);
//       test.testStringBuilderTime(100);
//       //连接 1000
//       test.testStringTime(1000);
//       test.testStringBufferTime(1000);
//       test.testStringBuilderTime(1000);
//      //连接 5000
//       test.testStringTime(5000);
//       test.testStringBufferTime(5000);
```



```
//      test.testStringBuilderTime(5000);
// //连接 10000
//      test.testStringTime(10000);
//      test.testStringBufferTime(10000);
//      test.testStringBuilderTime(10000);
// //连接 20000
//      test.testStringTime(20000);
//      test.testStringBufferTime(20000);
//      test.testStringBuilderTime(20000);
    }
    /**
     *测试 String 连接字符串的时间
     */
    public void testStringTime(int n){
       long start = System.currentTimeMillis();
       String a = "";
       for(int k=0;k<n;k++ ){
          a += " " + k;
       }
       long end = System.currentTimeMillis();
       long time = end - start;
       System.out.println("//////////////////////连接"+n+"次" );
       System.out.println("String time "+n +":"+ time);
       //System.out.println("String str:" + str);
    }
    /**
     *测试 StringBuffer 连接字符串的时间
     */
    public void testStringBufferTime(int n){
       long start = System.currentTimeMillis();
       StringBuffer b = new StringBuffer() ;
       for(int k=0;k<n;k++ ){
          b.append( " " + k );
       }
       long end = System.currentTimeMillis();
       long time = end - start;
       System.out.println("StringBuffer time "+n +":"+ time);
       //System.out.println("StringBuffer str:" + str);
    }
    /**
     *测试 StringBuilder 连接字符串的时间
     */
    public void testStringBuilderTime(int n){
       long start = System.currentTimeMillis();
       StringBuilder c = new StringBuilder() ;
       for(int k=0;k<n;k++ ){
          c.append( " " + k );
       }
       long end = System.currentTimeMillis();
       long time = end - start;
       System.out.println("StringBuilder time " +n +":"+ time);
```

```
        System.out.println("//////////////////////");
        //System.out.println("StringBuffer str:" + str);
    }
}
```

使用 Eclipse 运行上述代码，分别测试了当 n 等于 10、100、500、1000、5000、10000、20000 的时候，这三个对象连接字符串所花费的时间，统计结果如表 7-1 所示。

表 7-1　统计结果

连接次数(n)	所需时间(ms)		
	String	StringBuffer	StringBuilder
10	0	0	0
100	0	0	0
500	31	16	0
1000	63	31	16
5000	781	63	47
10000	7547	63	62
20000	62984	94	63

从表 7-1 的结果可以看出，为什么建议使用 StringBuffer 连接字符串的原因。在连接次数少的情况下，String 的低效率表现并不是很突出，但是一旦连接次数多的时候，性能影响是很大的。String 进行 2 万次字符串的连接，大约需要 1 分钟时间，而 StringBuffer 只需要 94 毫秒，相差接近 500 倍以上。而 StringBuffer 和 StringBuilder 差别并不大，StringBuilder 比 StringBuffer 稍微快点，我想是因为 StringBuffer 是线程安全的，StringBuilder 不是线程安全的，所以 StringBuffer 稍微慢点。

为什么 String 是如此之慢呢，请读者看下面的代码片段。

```
String result="";
result+="ok";
```

上述代码看上去好像没有什么问题，但是需要指出的是其性能很低，原因是 Java 中的类 String 是不可变的(immutable)，这段代码实际的工作过程是如何的呢？通过使用 javap 工具可以知道，其实上面的代码在编译成字节码时等同于下面的代码：

```
String result="";
StringBuffer temp=new StringBuffer();
temp.append(result);
temp.append("ok");
result=temp.toString();
```

短短的两个语句怎么变成这么多呢？问题的原因就在 String 类的不可变性上。而 Java 程序为了方便简单字符串的使用方式，对“+”操作符进行了重载，而这个重载的处理可能因此误导很多对 Java 中 String 的使用。所以，如果对字符串中的内容经常进行操作，特别是内容要修改时，那么建议使用 StringBuffer，如果最后需要 String，那么使用 StringBuffer 的方法 toString()。但是 StringBuilder 的实例用于多个线程是不安全的。如果

需要这样的同步，则建议使用 StringBuffer，因为 StringBuffer 是线程安全的。在大多数非多线程的开发中，为了提高效率，可以采用 StringBuilder 代替 StringBuffer，这样速度会更快。

7.2.3　及时释放不用的对象

在编写 Java 程序时，要养成及时释放不用的对象的习惯。例如当 a 不为空时，如下代码执行时将有两个对象存在于内存中。

```
a=new object()
```

而高效的写法是：

```
a=null;
a=new object();
```

我们需要及时将不用的对象设置成 null。

在 Java 中规定，因为内存溢出通常发生在构造函数中，所以在构造函数中，当使用某个变量时再 new，用完之后设置为 null。

另外，一次性加载所有图片会很容易造成内存峰值，此时也可以用 null 来解决，例如：

```
if(img==null){
Create…
}
```

请读者再看下面的两段代码，其中代码 2 是代码 1 执行速度的两倍。

代码 1：

```
String title=new String(“大家好”);
Title+=”欢迎”;
Title+=”阅读”
//会在栈中生成五个对象：“大家好”，“欢迎”，“阅读”，“大家好欢迎”，“大家好欢迎阅读”
```

代码 2：

```
StringBuffer title=new StringBuffer(“大家好”);
TItle.append(“欢迎”);
Title.append(“阅读”);
```

在 Java 程序中，StringBuffer 的构造器会创建一个默认大小(通常是 16)的字符数组。在使用过程中，如果超出这个大小，就会重新分配内存，创建一个更大的数组，并将原先的数组复制过来，再丢弃旧的数组。在大多数情况下，我们可以在创建 StringBuffer 的时候指定大小，这就避免了在容量不够的时候自动增长，以提高性能。

7.3　编写更高效的 Android 代码

因为基于 Android 平台的手持设备是嵌入式设备，而现代的手持设备不仅仅是一部电话那么简单，它还是一个小型的手持电脑，所以即使是最快的最高端的手持设备，也远远

比不上一台中等性能的桌面机。这就是为什么在编写 Android 程序时要时刻考虑执行效率的原因，因为这些系统不是想象中的那么快，并且你还要考虑它电池的续航能力。这就意味着没有多少剩余空间给你去浪费了，因此在我们编写 Android 程序的时候，要尽可能地使你的代码优化，从而提高效率。

7.3.1 避免建立对象

对于临时对象来说，每个线程分配池的垃圾回收器使得临时对象的创建只花出较小的代价，但分配内存总是比不分配内存花更多代价。如果在我们的一个用户界面循环中做分配对象操作，这样会产生一个定期的垃圾收集事件，使得界面会比较卡，影响用户体验。因此，应该避免创建对象实例。

当从原始的输入数据中提取字符串时，试着从原始字符串返回一个子字符串，而不是创建一份拷贝。你将会创建一个新的字符串对象，但是它和你的原始数据共享数据空间。

假如有一个返回字符串的方法，我们应该知道无论如何返回的结果是 StringBuffer，它可以改变函数的定义和执行，让函数直接返回而不是通过创建一个临时的对象。

一般来说，我们应该尽可能地避免创建短期的临时对象。越少的对象创建意味着越少的垃圾回收，这会提高程序的用户体验质量。

(1) 代码流程的优化

例如可以在代码设计流程中减少不必要的对象生成，看下面的演示代码：

```
Date myDate =new Date();
   if (requiredCondition) {
     // usemyDate
   }
```

我们可以将生成 Date()对象的语句放入 if 条件语句中，这样的话就可以有效减少不必要的对象生成。代码如下：

```
if (requiredCondition){
    Date myDate =new Date();
    // use myDate
}
```

这样只有在 if 条件成立的时候才创建对象，避免了不必要的创建对象工作。

(2) 对象在声明时的技巧

例如在使用 Vector 的过程中，经常声明一个 Vector 对象，但是不定义其初始大小。例如下面的演示代码：

```
Vector v = newVector();
```

这样做的弊端是 Vector 的内增长方法。当我们创建一个 Vector 对象时，当它的容量多于我们所声明的大小时，Vector 会默认先生成一个两倍大小的新的 Vector，然后再将原 Vector 中的内容拷贝一份到新 Vector。这样做的后果导致了垃圾回收时产生的性能问题。由此可见，除非万不得已，否则强烈建议在初始化时声明其大小，例如下面的演示代码：

```
Vector v = new Vector(40);
```

```
//or
Vector v = new Vector(40,25);
```

(3) 不要多次声明对象

除非有充分的理由，否则不要多次声明对象。例如下面的演示代码：

```
public class x{
    privateVector v = new Vector();
    public x() {
        v = new Vector();
    }
}
```

此时编译器会自动为构造函数生成如下代码：

```
public x() {
    v = new Vector();
    v = new Vector();
}
```

在默认情况下，任何事物都将被初始化为 Public 变量，初始化代码将被移动至构造函数中进行。所以，如果请不要在构造函数之外进行初始化，正确的声明方式如下：

```
public classx {
   privateVector v;
   public x() {
     v = new Vector();
    }
}
```

由此可见，在 Android 应用程序中如果没有必要就不应该创建对象实例。

- 当从原始的输入数据中提取字符串时，试着从原始字符串返回一个子字符串，而不是创建一份拷贝。你将会创建一个新的字符串对象，但是它和你的原始数据共享数据空间。
- 如果你已经有一个返回字符串的方法，你应该知道无论如何返回的结果是 StringBuffer，改变你的函数的定义和执行，让函数直接返回而不是通过创建一个临时的对象。

除此之外，还有一个比较激进的方法，就是把一个多维数组分割成几个平行的一维数组。

- 一个 Int 类型的数组要比一个 Integer 类型的数组好，但着同样也可以归纳于这样一个原则，两个 Int 类型的数组要比一个(int，int)对象数组的效率高得多。对于其他原始数据类型，这个原则同样适用。
- 当你需要创建一个包含一系列 Foo 和 Bar 对象的容器(container)时，两个平行的 Foo[]和 Bar[]要比一个(Foo,Bar)对象数组的效率高得多。这个例子也有一个例外，当你设计其他代码的接口 API 时，速度上的一点损失就不用考虑了。但是在我们的代码里面，应该尽可能地编写高效代码。

由此可以总结到，我们应该尽可能地避免创建短期的临时对象。越少的对象创建意味着越少的垃圾回收，这会提高程序的用户体验质量。

7.3.2 优化方法调用代码

(1) 使用自身方法

当处理字符串的时候，不要犹豫，要尽可能多地使用诸如 String.indexOf()、String.lastIndexOf()等对象自身带有的方法。因为这些方法是用 C/C++实现的，要比在一个 Java 循环中做同样的事情快 10～100 倍。

(2) 使用虚拟优于使用接口

假设你有一个 HashMap 对象，则可以声明它是一个 HashMap 或只是一个 Map，下面是演示代码。

```
Map myMap1 = new HashMap();
HashMap myMap2 = new HashMap();
```

这样究竟哪一个更好呢？一般来说，明智的做法是使用 Map，因为它能够允许我们改变 Map 接口执行上面的任何东西，但是这种“明智”的方法只是适用于常规的编程，对于嵌入式系统并不适合。相对于通过具体的引用进行虚拟函数的调用，通过接口引用来调用会花费两倍以上的时间。

如果选择使用 HashMap，因为它更适合于我们的编程，那么如果使用 Map 会毫无价值。假设有一个能重构我们代码的集成编码环境，那么调用 Map 将没有任何用处，即使我们不确定程序从哪儿开头。同样，public 的 API 是一个例外，一个好的 API 的价值往往大于执行效率上的那点损失。

(3) 使用静态优于使用虚拟

如果没有必要去访问对象的外部，那么使我们的方法成为静态方法。它会被更快地调用，因为它不需要一个虚拟函数导向表。这同时也是一个很好的实践，因为它告诉我们如何区分方法的性质(signature)。调用这个方法不会改变对象的状态。

(4) 尽可能避免使用内在的 Get、Set 方法

像 C++之类的编程语言，通常会使用 Get 方法(例如 i=getCount())去取代直接访问这个属性(i=mCount)。这在 C++编程里面是一个很好的习惯，因为编译器会把访问方式设置为 Inline，并且如果想约束或调试属性访问，只需在任何时候添加一些代码即可。

但是在 Android 编程中，这是一个很不好的主意。虚方法的调用会产生很多代价，比实例属性查询的代价还要多。我们应该在外部调用时使用 Get 和 Set 函数，但是在内部调用时，我们应该直接调用。

(5) 要使用 getBytes()

在将 String 转化成 bytes 的过程中，不要使用 getBytes()函数。例如，当我们在处理 HTTP 字符串时，在绝大多数情况下，它们都是 ASCII 码。getBytes()函数可以处理几乎所有字符的编码问题。但是这种能力在 HTTP 事务处理中似乎并不必要。你可以创建你自己的方法去处理仅仅一种 ASCII 码。

看下面的演示代码：

```
public static void mySimpleTokenizer(String s, String delimiter)
 {
```

```
     String sub = null;
     int i =0;
     int j =s.indexOf(delimiter);  // First substring
     while( j >= 0)
    {
         sub = s.substring(i,j);
         i = j + 1;
         j = s.indexOf(delimiter, i);   // Rest of substrings
     }
      sub = s.substring(i); // Last substring
 }
// 现在就可以直接调用了
byte[] b =getAsciiBytes(s);
```

(6) 尽量避免使用 InetAddress.getHostAddress()

因为 InetAddress.getHostAddress()包含了许多操作，所以它会生成许多中间字符串来返回主机地址，这大大增加了 Android 应用程序在时间上的负担。

(7) 尽量避免使用 DatagramPacket.getSocketAddress()

DatagramPacket.getSocketAddress()也包含了许多操作，调用时函数内部调用会尝试返回其主机名，这大大增加了 Android 应用程序在时间上的负担。如果只是要获得 Android 应用程序数据包的 IP 地址，可以用 DatagramPacket.getAddress().getHostAddress()函数代替。

7.3.3　优化代码变量

(1) StringBuffer 的使用

这一条和 Java 中的优化规则一样，例如当需要对一组 String 进行连接时，请不要使用下面的代码。

```
String str=  "Welcome"+ "to" + "our" + "site";
```

而应当写成：

```
StringBuffer sb  =  new StringBuffer(50);
 sb.append("Welcome");
 sb.append("To");
 sb.append("our");
 sb.append("site");
```

如果知道 StringBuffer 的最大长度，请使用这个数字。例如：在上面的代码中，StringBuffer 的最大长度设置为 50，这使得在使用 StringBuffer 的过程中，不需要考虑自增长问题。这样就不需要再去为 StringBuffer 分配新的内存，而导致垃圾回收器回收旧的内存。当然也不要分配过于大的、不必要的内存。

(2) 声明 Final 常量

我们可以看看下面一个类顶部的声明：

```
static int intVal = 42;
static String strVal = "Hello, world!";
static int intVal = 42;
static String strVal = "Hello, world!";
```

当第一次使用一个类时，编译器会调用一个类初始化方法：<clinit>。这个方法将 42 存入变量 intVal 中，并且为 strVal 在类文件字符串常量表中提取一个引用，当这些值在后面引用时，就会直接访问。我们可以用关键字“final”来改进代码：

```
static final int intVal = 42;
static final String strVal = "Hello, world!";
static final int intVal = 42;
static final String strVal = "Hello, world!";
```

这样此类将不会调用<clinit>方法，因为这些常量直接写入了类文件静态属性初始化中，这个初始化直接由虚拟机来处理。当代码访问 intVal 时，将会使用 Integer 类型的 42；当访问 strVal 时，将会使用相对节省的“字符串常量”来替代一个属性调用。

如果将一个类或者方法声明为“final”，并不会带来任何执行上的好处，它能够进行一定的最优化处理。例如，如果编译器知道一个 Get 方法不能被子类重载，那么它就把该函数设置成 Inline。

同时，我们也可以把本地变量声明为 final 变量，但是这是毫无意义的。作为一个本地变量，使用 final 只能使代码更加清晰(或者你不得不用，在匿名访问内联类时)。

(3) 避免使用列举类型

列举类型非常好用，当考虑到大小和速度的时候，就会显得代价很高，例如下面的演示代码：

```
public class Foo {
public enum Shrubbery {
GROUND, CRAWLING, HANGING
}
}
public class Foo {
public enum Shrubbery {
GROUND, CRAWLING, HANGING
}
}
```

通过上述代码，会转变成为一个 900 字节的 class 文件(Foo$Shrubbery.class)。当第一次使用时，类的初始化要调用方法去描述列举的每一项，每一个对象都要有它自身的静态空间，整个被储存在一个数组里面(一个叫作“$VALUE”的静态数组)。那是一大堆的代码和数据，仅仅是为了三个整数值。

(4) 避免使用枚举

枚举变量非常方便，但是这是以牺牲执行的速度和大幅增加文件体积为前提的。例如下面的演示代码：

```
public class Foo {
 public enum Shrubbery {
 GROUND, CRAWLING, HANGING
 }
 }
```

上述代码会产生一个 900 字节的.class 文件(Foo$Shubbery.class)。在它被首次调用时，

这个类会调用初始化方法来准备每个枚举变量。每个枚举项都会被声明成一个静态变量，并被赋值。然后将这些静态变量放在一个名为$VALUES 的静态数组变量中。而这么一大堆代码，仅仅是为了使用三个整数。

这样下面的代码会引起一个对静态变量的引用，如果这个静态变量是 final int，那么编译器会直接内联这个常数。

```
Shrubbery shrub = Shrubbery.GROUND;
```

一方面说，使用枚举变量可以让你的 API 更出色，并能提供编译时的检查。所以在通常的时候你毫无疑问应该为公共 API 选择枚举变量。但是当性能方面有所限制的时候，你就应该避免这种做法了。在有些情况下，使用方法 ordinal()获取枚举变量的整数值会更好一些，举例来说，如果将：

```
for (int n = 0; n < list.size(); n++) {
 if (list.items[n].e == MyEnum.VAL X) // do stuff 1
 else if (list.items[n].e == MyEnum.VAL Y) // do stuff 2
 }
```

替换为：

```
int valX = MyEnum.VAL_X.ordinal();
int valY = MyEnum.VAL Y.ordinal();
int count = list.size();
MyItem items = list.items();
for (int n = 0; n < count; n++) {
 int valItem = items[n].e.ordinal();
 if (valItem == valX) // do stuff 1
 else if (valItem == valY) // do stuff 2
 }
```

这样会使性能得到一些改善，但这并不是最终的解决之道。

如果将与内部类一同使用的变量声明在包范围内，请看下面的类定义：

```
public class Foo {
 private int mValue;
 public void run() {
 Inner in = new Inner();
 mValue = 27;
 in.stuff();
 }
 private void doStuff(int value) {
  System.out.println("Value is " + value);
 }
 private class Inner {
 void stuff() {
 Foo.this.doStuff(Foo.this.mValue);
 }
 }
 }
```

这其中的关键是，我们定义了一个内部类(Foo$Inner)，它需要访问外部类的私有域变

量和函数。这是合法的，并且会打印出我们希望的结果：

```
Value is 27
```

但是问题是在技术上来讲，Foo$Inner 是一个完全独立的类，它要直接访问 Foo 的私有成员是非法的。要跨越这个鸿沟，编译器需要生成一组方法：

```
static int Foo.access$100(Foo foo) {
 return foo.mValue;
 }
 static void Foo.access$200(Foo foo, int value) {
 foo.doStuff(value);
 }
```

当内部类在每次访问 mValue 和 doStuff 方法时，都会调用这些静态方法。也就是说，上面的代码说明了一个问题，我们是在通过接口方法访问这些成员变量和函数而不是直接调用它们。前面我们已经说过，使用接口方法(getter、setter)比直接访问速度要慢。所以这个例子就是在特定语法下面产生的一个“隐性的”性能障碍。

通过将内部类访问的变量和函数声明由私有范围改为包范围，我们可以避免这个问题。这样做可以让代码运行更快，并且避免产生额外的静态方法。

7.3.4 优化代码过程

(1) 慎重使用增强型 for 循环语句

增强型 for 循环也就是我们常说的“for-each 循环”，经常用于 iterable 接口的继承收集接口上面。在这些对象里面，一个 iterator 被分配给对象去调用它的 hasNext()和 next()方法。虽然如此，下面的演示代码还是给出了一个可以接受的增强型 for 循环的例子。

```
public class Foo {
int mSplat;
static Foo mArray[] = new Foo[27];
public static void zero() {
int sum = 0;
for (int i = 0; i < mArray.length; i++) {
sum += mArray[i].mSplat;
}
}
public static void one() {
int sum = 0;
Foo[] localArray = mArray;
int len = localArray.length;
for (int i = 0; i < len; i++) {
sum += localArray[i].mSplat;
}
}
public static void two() {
int sum = 0;
for (Foo a: mArray) {
sum += a.mSplat;
```



```
}
}
}
public class Foo {
int mSplat;
static Foo mArray[] = new Foo[27];
public static void zero() {
int sum = 0;
for (int i = 0; i < mArray.length; i++) {
sum += mArray[i].mSplat;
}
}
public static void one() {
int sum = 0;
Foo[] localArray = mArray;
int len = localArray.length;
for (int i = 0; i < len; i++) {
sum += localArray[i].mSplat;
}
}
public static void two() {
int sum = 0;
for (Foo a: mArray) {
sum += a.mSplat;
}
}
}
```

对上述代码的具体说明如下。

- 函数 zero()：在每一次的循环中重新得到静态属性两次，获得数组长度一次。
- 函数 one()：把所有的东西都变为本地变量，避免类查找属性调用。
- 函数 two()：使用 Java 语言的 1.5 版本中的 for 循环语句，编译产生的源代码考虑到了拷贝数组的引用和数组的长度到本地变量，是遍历数组比较好的方法，它在主循环中确实产生了一个额外的载入和储存过程(显然保存了“a”)，相比函数 one()来说，它有一点减慢和 4 字节的增长。

由此可以得到，增强的 for 循环在数组里面表现得很好，但是当和 Iterable 对象一起使用时要谨慎，因为这里多了一个对象的创建。

(2) 通过内联类使用包空间

请看如下代码中对类的声明：

```
public class Foo {
private int mValue;
public void run() {
Inner in = new Inner();
mValue = 27;
in.stuff();
}
private void doStuff(int value) {
```

```
System.out.println("Value is " + value);
}
private class Inner {
void stuff() {
Foo.this.doStuff(Foo.this.mValue);
}
}
}
public class Foo {
private int mValue;
public void run() {
Inner in = new Inner();
mValue = 27; in.stuff();
}
private void doStuff(int value) {
System.out.println("Value is " + value);
}
private class Inner {
void stuff() {
Foo.this.doStuff(Foo.this.mValue);
}
}
}
```

在上述代码中，需要注意我们定义了一个内联类，它调用了外部类的私有方法和私有属性。这是一个合法的调用，代码应该会显示：

```
Value is 27
```

但是问题是，Foo$Inner 在理论上(后台运行上)是应该是一个完全独立的类，它违规调用了 Foo 的私有成员。为了弥补这个缺陷，编译器产生了一对合成的方法：

```
/*package*/
static int Foo.access$100(Foo foo) {
return foo.mValue;
}
/*package*/
static void Foo.access$200(Foo foo, int value) {
foo.doStuff(value);
}
/*package*/
static int Foo.access$100(Foo foo) {
return foo.mValue;
}
/*package*/
static void Foo.access$200(Foo foo, int value) {
foo.doStuff(value);
}
```

这样当内联类需要从外部访问 mValue 和调用 doStuff 时，内联类就会调用这些静态的方法，这说明我们不是直接访问类成员，而是通过公共的方法访问的。前面曾经提到过间接访问要比直接访问慢，因此这是一个按语言习惯无形执行的例子。



如果让拥有包空间的内联类直接声明需要访问的属性和方法，我们就可以避免这个问题，这里是包空间而不是私有空间。这样不但运行的更快，并且去除了生成函数前面东西。但是不幸的是，它同时也意味着该属性也能够被相同包下面的其他的类直接访问，这违反了标准的面向对象的使所有属性私有的原则。同样，如果是设计公共的 API 你就要仔细地考虑这种优化的用法。

(3) 避免浮点类型的使用

在奔腾 CPU 发布之前，游戏程序员都尽可能地使用 Integer 类型的数学函数，这是很正常的。因为在奔腾处理器里面，浮点数的处理是一个突出的特点，并且浮点数与整数的交互使用相比单独使用整数来说，前者会使你的游戏运行得更快，一般的在桌面电脑上面我们可以自由地使用浮点数。

但是不幸的是，嵌入式的处理器通常并不支持浮点数的处理，因此所有的“float”和“double”操作都是通过软件进行的，一些基本的浮点数的操作就需要花费毫秒级的时间。并且同时即使是整数，一些芯片也只有乘法而没有除法。在这些情况下，整数的除法和取模操作都是通过软件实现的。当你创建一个 Hash 表或者进行大量的数学运算时，这都是你要考虑的。

(4) 避免在条件判定语句中重复调用函数

请读者看下面的演示代码：

```
for (int i=0 ; i < s.length; i++) {
   charc c =s.charAt(i);
}
```

应该写成下面的形式，因为这样可以减少时间开销。

```
int j =str.length();
for (int i =0 ; i < j; i++) {
   charc c =s.charAt(i);
}
```

7.3.5　提高 Cursor 查询数据的性能

在 Android 系统中，查询数据的功能是通过类 Cursor 实现的，使用方法 sqlitedatabase.query()就能得到 Cursor 对象，Cursor 对象代表每行的集合。当解析 Cursor 对象时，如果只是解析一行，可通过方法 moveToFirst()定位到第一行。当再解析时，如果是多于一行的，则可以在 while 循环条件里加上 moveToNext()定位后再解析。

当 Cursor 中的数据只有一行时，代码优化工作会比较省事，我们基本上不用担心会因代码不好影响性能。但是当里面的数据量很多时，如果没有优化代码，则对解析的速度会带来很大的影响。

在定位后解析 cursor 时，我们一般的做法是首先通过方法 getColumnIndex(String columnName)获得列的索引值，然后再通过列的索引值获得对应的数据。就像如下代码中这样，实现了对联系人部分数据的解析。

```
while(cursor.moveToNext){
   String name = cursor.getString(cursor.getColumnIndex(People.Name));
```

```
    String phoneNo =
cursor.getString(cursor.getColumnIndex(People.Number));
}
```

上述代码没有任何错误，最后解析出来的结果也是完全正确。但是这段代码其实写得很差，当在一定量的联系人数据时，运行速度会相当慢。我们可以用这种代码写出来的程序与系统自带的通讯录(或者 QQ 通讯录)来比较一下三百多联系人的数据，就知道有多慢了。

在进行优化时，我们只需稍稍地做一下改变，执行速度将会有质的提升。改造如下：

```
int nameIndex = cursor.getColumnIndex(People.Name);
int numberIndex = cursor.getColumnIndex(People.NUMBER);
while(cursor.moveToNext){
    String name = cursor.getString(nameIndex);
    String phoneNo = cursor.getString(numberIndex);
}
```

这样经过改造以后，可以再测试一下三百多条联系人的数据，此时会基本接近系统联系人的速度(前提是在不加载头像的情况，头像要用另外一套机制去解决，在此不做讨论)。

经过上述两段代码的讨论，相信大家都应该知道是如何优化的了——就是把列的索引值获取提取到循环前面去。别小看这一点点修改，它能够帮我们的大忙。这样修改的好处就是，能让程序避免重复去获得这些列的索引值，使程序的运行效率更高，特别是在写联系人的程序时很有效。读者可以举一反三，在我们平时写代码时很多逻辑都可以这样去优化。最后，记得解析完后要关闭 Cursor。

7.3.6 编码中尽量使用 ContentProvider 共享数据

众所周知，在 Android 应用中的最通用数据库是 SQLite。但是 Google 为了给我们简化操作，可以不用经常编写容易出错的 SQL 语句，而是可以直接通过 ContentProvider 来封装数据的 query 查询、添加 insert、删除 delete 和更新 update，而无须用复杂的 SQLite，提高了程序运行效率。

接下来以 Android 系统的 SDK 中的例子，来讲解使用 ContentProvider 共享数据的好处。

```
public class NotePadProvider extends ContentProvider {
private static final String TAG = "NotePadProvider";
private static final String DATABASE NAME = "note pad.db"; //数据库存储文件名，包含了.db 后缀
private static final int DATABASE_VERSION = 2; //数据库版本号，这个是自己定义的，未来扩展数据库时自己可以方便地定义升级规则
private static final String NOTES_TABLE_NAME = "notes"; //表名
private static HashMap sNotesProjectionMap; //常规的 Notes
private static HashMap sLiveFolderProjectionMap; //LiveFoder 内容
private static final int NOTES = 1;
private static final int NOTE_ID = 2;
private static final int LIVE FOLDER NOTES = 3;
private static final UriMatcher sUriMatcher; //这里提示大家，通常我们操作数据库的 Uri 比如 content://android123/cwj/1103 这样的 Uri 均通过 UriMatcher 注册并识
```



```
别的。
private static class DatabaseHelper extends SQLiteOpenHelper { //数据库辅
助子类
DatabaseHelper(Context context) {
super(context, DATABASE NAME, null, DATABASE VERSION);
}
@Override
public void onCreate(SQLiteDatabase db) { //首次生成数据库，执行 SQL 命令创建
一个表
db.execSQL("CREATE TABLE " + NOTES TABLE NAME + " ("
+ Notes. ID + " INTEGER PRIMARY KEY,"
+ Notes.TITLE + " TEXT,"
+ Notes.NOTE + " TEXT,"
+ Notes.CREATED DATE + " INTEGER,"
+ Notes.MODIFIED DATE + " INTEGER"
+ ");");
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{ //刚来数据的版本，就是为了定义我们如果未来数据库需要扩展，帮助用户识别并根据规则自动
升级数据库文件
Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
+ newVersion + ", which will destroy all old data");
db.execSQL("DROP TABLE IF EXISTS notes"); //由于这里没有做细节处理，如果有新版
本，删除老的表，我们未来不能这样处理，这仅仅是 Google 的例子而已所以删除老版本数据
onCreate(db);
}
}
private DatabaseHelper mOpenHelper;
@Override
public boolean onCreate() { //这里重写 ContentProvider 的 onCreate 方法做一些
初始化操作
mOpenHelper = new DatabaseHelper(getContext());
return true;
}
//有关数据库的查询操作，Android 的 SQLite 提供了一个 SQLiteQueryBuilder 方法再次将
SQL 命令封装了下，单独分离出表名，排序方法等
@Override
public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs,
String sortOrder) {
SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
qb.setTables(NOTES TABLE NAME);
switch (sUriMatcher.match(uri)) {
case NOTES:
qb.setProjectionMap(sNotesProjectionMap);
break;
case NOTE ID:
qb.setProjectionMap(sNotesProjectionMap);
qb.appendWhere(Notes. ID + "=" + uri.getPathSegments().get(1));
break;
```

```
case LIVE FOLDER NOTES:
qb.setProjectionMap(sLiveFolderProjectionMap);
break;
default:
throw new IllegalArgumentException("Unknown URI " + uri);
}
String orderBy;
if (TextUtils.isEmpty(sortOrder)) {
orderBy = NotePad.Notes.DEFAULT SORT ORDER;
} else {
orderBy = sortOrder;
}
SQLiteDatabase db = mOpenHelper.getReadableDatabase();
Cursor c = qb.query(db, projection, selection, selectionArgs, null, null,
orderBy);
c.setNotificationUri(getContext().getContentResolver(), uri);
return c;
}
@Override
public String getType(Uri uri) {
switch (sUriMatcher.match(uri)) {
case NOTES:
case LIVE FOLDER NOTES:
return Notes.CONTENT TYPE;
case NOTE ID:
return Notes.CONTENT ITEM TYPE;
default:
throw new IllegalArgumentException("Unknown URI " + uri);
}
}
```

有关数据的插入操作，只需重写 ContentProvider 的方法 insert()即可。

```
@Override
public Uri insert(Uri uri, ContentValues initialValues) {
if (sUriMatcher.match(uri) != NOTES) {
throw new IllegalArgumentException("Unknown URI " + uri);
}
ContentValues values;
if (initialValues != null) {
values = new ContentValues(initialValues);
} else {
values = new ContentValues();
}
Long now = Long.valueOf(System.currentTimeMillis());
if (values.containsKey(NotePad.Notes.CREATED DATE) == false) {
values.put(NotePad.Notes.CREATED DATE, now);
}
if (values.containsKey(NotePad.Notes.MODIFIED DATE) == false) {
values.put(NotePad.Notes.MODIFIED DATE, now);
}
```



```
if (values.containsKey(NotePad.Notes.TITLE) == false) {
Resources r = Resources.getSystem();
values.put(NotePad.Notes.TITLE, r.getString(android.R.string.untitled));
}
if (values.containsKey(NotePad.Notes.NOTE) == false) {
values.put(NotePad.Notes.NOTE, "");
}
SQLiteDatabase db = mOpenHelper.getWritableDatabase();
long rowId = db.insert(NOTES TABLE NAME, Notes.NOTE, values);
if (rowId > 0) {
Uri noteUri = ContentUris.withAppendedId(NotePad.Notes.CONTENT URI,
rowId);
getContext().getContentResolver().notifyChange(noteUri, null); //通知数据
库内容有改变
return noteUri;
}
throw new SQLException("Failed to insert row into " + uri);
}
@Override
public int delete(Uri uri, String where, String[] whereArgs) {
SQLiteDatabase db = mOpenHelper.getWritableDatabase();
int count;
switch (sUriMatcher.match(uri)) {
case NOTES:
count = db.delete(NOTES TABLE NAME, where, whereArgs);
break;
case NOTE ID:
String noteId = uri.getPathSegments().get(1);
count = db.delete(NOTES TABLE NAME, Notes. ID + "=" + noteId
+ (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""), whereArgs);
break;
default:
throw new IllegalArgumentException("Unknown URI " + uri);
}
getContext().getContentResolver().notifyChange(uri, null);
return count;
}
@Override
public int update(Uri uri, ContentValues values, String where, String[]
whereArgs) {
SQLiteDatabase db = mOpenHelper.getWritableDatabase();
int count;
switch (sUriMatcher.match(uri)) {
case NOTES:
count = db.update(NOTES TABLE NAME, values, where, whereArgs);
break;
case NOTE ID:
String noteId = uri.getPathSegments().get(1);
count = db.update(NOTES TABLE NAME, values, Notes. ID + "=" + noteId
+ (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""), whereArgs);
break;
```

```
default:
throw new IllegalArgumentException("Unknown URI " + uri);
}
getContext().getContentResolver().notifyChange(uri, null);
return count;
}
```

最后我们需要在构造奔雷时就监听 Uri，如果处理的 Uri 需要其他程序获知，需要在 Androidmanifest.xml 文件中显式地导出 provider 的 Uri 定义。

```
static {
sUriMatcher = new UriMatcher(UriMatcher.NO MATCH);
sUriMatcher.addURI(NotePad.AUTHORITY, "notes", NOTES);
sUriMatcher.addURI(NotePad.AUTHORITY, "notes/#", NOTE ID);
sUriMatcher.addURI(NotePad.AUTHORITY, "live folders/notes",
LIVE FOLDER NOTES);
sNotesProjectionMap = new HashMap();
sNotesProjectionMap.put(Notes. ID, Notes. ID);
sNotesProjectionMap.put(Notes.TITLE, Notes.TITLE);
sNotesProjectionMap.put(Notes.NOTE, Notes.NOTE);
sNotesProjectionMap.put(Notes.CREATED DATE, Notes.CREATED DATE);
sNotesProjectionMap.put(Notes.MODIFIED DATE, Notes.MODIFIED DATE);
// Support for Live Folders.
sLiveFolderProjectionMap = new HashMap();
sLiveFolderProjectionMap.put(LiveFolders. ID, Notes. ID + " AS " +
LiveFolders. ID);
sLiveFolderProjectionMap.put(LiveFolders.NAME, Notes.TITLE + " AS " +
LiveFolders.NAME);
// Add more columns here for more robust Live Folders.
}
}
```

要想开发出高效的 ContentProvider 存储应用，就要尽可能地减少 SQL 语句的编写在外部操作，封装成方法，这样有关 SQL 语言的执行在 DatabaseHelper 中也被简化和分离出来了，而 SQL 语句主要是体现在选择表的字段、where 这样的条件限定语句，这大大减少了我们日常的开发工作量，从而实现了优化工作。

7.4 Android 控件的性能优化

控件是 Android 应用中的常用组成元素，通过使用控件，我们无须编写很多代码，只需直接调用控件即可实现我们需要的功能。本节简要讲解优化 Android 控件的基本知识。

7.4.1 ListView 控件的代码优化

ListView 是 Android 应用中的最常用控件之一，能够实现列表显示数据功能。在本节的内容中，将通过具体的演示代码来测试 ListView 控件的性能。

(1) 首先看如下测试代码：

```
private TestAdapter mAdapter;
    private String[] mArrData;
    private TextView mTV;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mTV = (TextView) findViewById(R.id.tvShow);
        mArrData = new String[1000];
        for (int i = 0; i < 1000; i++) {
            mArrData[i] = "Google IO Adapter" + i;
        }
        mAdapter = new TestAdapter(this, mArrData);
        ((ListView) findViewById(android.R.id.list)).setAdapter(mAdapter);
    }
```

通过上述代码模拟了 1000 条数据，设置了 TestAdapter 继承于 BaseAdapter。

(2) 然后开始具体测试，手动滑动 ListView，当位置变量 position 到 50 时往回滑动，充分利用 convertView 不等于 null 的代码段。为了实现具体测试功能，我们用如下三种方案进行测试。

① 方案 1：把 item 子元素分别改为 4 个和 10 个，这样做的目的是使效果更加明显。

```
private int count = 0;
        private long sum = 0L;
        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
            //开始计时
            long startTime = System.nanoTime();

            if (convertView == null) {
                convertView = mInflater.inflate(R.layout.list item icon text,
                        null);
            }
            ((ImageView)
convertView.findViewById(R.id.icon1)).setImageResource(R.drawable.icon);
            ((TextView)
convertView.findViewById(R.id.text1)).setText(mData[position]);
            ((ImageView)
convertView.findViewById(R.id.icon2)).setImageResource(R.drawable.icon);
            ((TextView)
convertView.findViewById(R.id.text2)).setText(mData[position]);

            //停止计时
            long endTime = System.nanoTime();
            //计算耗时
            long val = (endTime - startTime) / 1000L;
            Log.e("Test", "Position:" + position + ":" + val);
            if (count < 100) {
```

```
            if (val < 1000L) {
                sum += val;
                count++;
            }
        } else
            mTV.setText(String.valueOf(sum / 100L));//显示统计结果
        return convertView;
    }
```

上述方案的测试结果如表 7-2 所示。

表 7-2　测试结果(单位：微秒除以 1000)

次　数	4 个子元素	10 个子元素
第一次	366	723
第二次	356	689
第三次	371	692
第四次	356	696
第五次	371	662

② 方案 2：把 item 子元素分别改为 4 个和 10 个。

```
private int count = 0;
    private long sum = 0L;
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        // 开始计时
        long startTime = System.nanoTime();
        ViewHolder holder;
        if (convertView == null) {
            convertView = mInflater.inflate(R.layout.list item icon text,
                    null);
            holder = new ViewHolder();
            holder.icon1 = (ImageView) convertView.findViewById(R.id.icon1);
            holder.text1 = (TextView) convertView.findViewById(R.id.text1);
            holder.icon2 = (ImageView) convertView.findViewById(R.id.icon2);
            holder.text2 = (TextView) convertView.findViewById(R.id.text2);
            convertView.setTag(holder);
        }
        else{
            holder = (ViewHolder)convertView.getTag();
        }
        holder.icon1.setImageResource(R.drawable.icon);
        holder.text1.setText(mData[position]);
        holder.icon2 .setImageResource(R.drawable.icon);
        holder.text2.setText(mData[position]);
        // 停止计时
        long endTime = System.nanoTime();
        // 计算耗时
        long val = (endTime - startTime) / 1000L;
```



```
            Log.e("Test", "Position:" + position + ":" + val);
            if (count < 100) {
                if (val < 1000L) {
                    sum += val;
                    count++;
                }
            } else
                mTV.setText(String.valueOf(sum / 100L));// 显示统计结果
            return convertView;
        }
    }
    static class ViewHolder {
        TextView text1;
        ImageView icon1;
        TextView text2;
        ImageView icon2;
    }
```

上述方案的测试结果如表 7-3 所示。

表 7-3 测试结果(单位：微秒除以 1000)

次 数	4 个子元素	10 个子元素
第一次	311	417
第二次	291	441
第三次	302	462
第四次	286	444
第五次	299	436

③ 方案 3：原理是减少 findViewById 次数，并顺便测试不使用静态内部类情况下的性能。

```
@Override
        public View getView(int position, View convertView, ViewGroup parent) {
            // 开始计时
            long startTime = System.nanoTime();
            if (convertView == null) {
                convertView =
mInflater.inflate(R.layout.list_item_icon_text, null);
                convertView.setTag(R.id.icon1,
convertView.findViewById(R.id.icon1));
                convertView.setTag(R.id.text1,
convertView.findViewById(R.id.text1));
                convertView.setTag(R.id.icon2,
convertView.findViewById(R.id.icon2));
                convertView.setTag(R.id.text2,
convertView.findViewById(R.id.text2));
            }
            ((ImageView) convertView.getTag(R.id.icon1)).setImageResource
(R.drawable.icon);
```

```
            ((ImageView) convertView.getTag(R.id.icon2)).setImageResource
(R.drawable.icon);
            ((TextView) convertView.getTag(R.id.text1)).setText (mData[position]);
            ((TextView) convertView.getTag(R.id.text2)).setText (mData[position]);
            // 停止计时
            long endTime = System.nanoTime();
            // 计算耗时
            long val = (endTime - startTime) / 1000L;
            Log.e("Test", "Position:" + position + ":" + val);
            if (count < 100) {
                if (val < 1000L) {
                    sum += val;
                    count++;
                }
            } else
                mTV.setText(String.valueOf(sum / 100L) + ":" +
nullcount);// 显示统计结果
            return convertView;
        }
```

上述方案的测试结果如下(单位：微秒除以 1000)

```
第一次：450
第二次：467
第三次：472
第四次：451
第五次：441
```

执行上述三种方案，经过测试后发现，只有第一屏(可视范围)调用 getView 所消耗的时间远远多于后面的，通过对 convertView == null 内代码监控后发现也是同样的结果。也就是说，ListView 仅仅缓存了可视范围内的 View，随后的滚动都是对这些 View 进行数据更新。无论有有多少数据，ListView 都只用 ArrayList 缓存可视范围内的 View，这样保证了性能，也造成了我以为 ListView 只缓存 View 结构不缓存数据的假象。这也是上述优化方案 1 比方案 2 高很多的原因，那么剩下的工作也就只有 findViewById 比较耗时了。

根据上述原理可以推到出，如下代码运行后会发现滚动时会重复显示第一屏的数据。

```
if (convertView == null) {
                convertView = mInflater.inflate(R.layout.list item icon text,
null);
                ((ImageView) convertView.findViewById(R.id.icon1)).
setImageResource(R.drawable.icon);
                ((TextView) convertView.findViewById(R.id.text1)).setText
(mData[position]);
                ((ImageView) convertView.findViewById(R.id.icon2)).
setImageResource(R.drawable.icon);
                ((TextView) convertView.findViewById(R.id.text2)).
setText(mData[position]);
            }
            else
                return convertView;
```



在上述代码中，因为子控件里的事件是同一个控件，所以也可以直接放到 convertView == null 代码块内部，如果需要交互数据比如 position，可以通过 tag 方式来设置并获取当前数据。

在上述三种方案中，推荐如果只是一般的应用(一般指子控件不多)，无须都是用静态内部类来优化，使用第 2 种方案即可；反之，如果是对性能要求较高时可采用。此外需要提醒的是，这里也是用空间换时间的做法，View 本身因为 setTag 而会占用更多的内存，还会增加代码量；而 findViewById 会临时消耗更多的内存，所以不可盲目使用，需要依实际情况而定。至于方案 3，原理是减少 findViewById 次数，但是从测试结果来看效果并不理想，所以在此不再做进一步的测试和讨论。

7.4.2　Adapter(适配器)优化

Adapter 和 ListView 控件密切相关，Adapter 的作用就是 ListView 界面与数据之间的桥梁，当列表里的每一项显示到页面时，都会调用 Adapter 的 getView()方法返回一个 View。Adapter 与 View 的连接主要依靠 getView 这个方法返回我们需要的自定义 view。ListView 是 Android app 中一个最最最常用的控件了，所以如何让 ListView 流畅运行，获取良好的用户体验是非常重要的。对 ListView 优化就是对 Adapter 中的 getView 方法进行优化。请看 Google IO 大会给 Adapter 的优化建议：

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    Log.d("MyAdapter", "Position:" + position + "---"
          + String.valueOf(System.currentTimeMillis()));
    ViewHolder holder;
    if (convertView == null) {
        final LayoutInflater inflater = (LayoutInflater) mContext
               .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        convertView = inflater.inflate(R.layout.list_item_icon_text, null);
       holder = new ViewHolder();
        holder.icon = (ImageView) convertView.findViewById(R.id.icon);
        holder.text = (TextView) convertView.findViewById(R.id.text);
        convertView.setTag(holder);
    } else {
        holder = (ViewHolder) convertView.getTag();
    }
    holder.icon.setImageResource(R.drawable.icon);
    holder.text.setText(mData[position]);
    return convertView;
}
static class ViewHolder {
    ImageView icon;
    TextView text;
 }
```

经过尝试以后，发现 ListView 确实流畅了许多。

而下面是 Google IO 不建议的做法：

```
@Override
 public View getView(int position, View convertView, ViewGroup parent) {
     Log.d("MyAdapter", "Position:" + position + "---"
            + String.valueOf(System.currentTimeMillis()));
        final LayoutInflater inflater = (LayoutInflater) mContext
               .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        View v = inflater.inflate(R.layout.list_item_icon_text, null);
        ((ImageView) v.findViewById(R.id.icon)).setImageResource(R.drawable.icon);
        ((TextView) v.findViewById(R.id.text)).setText(mData[position]);
        return v;
 }
```

针对上述两种方案，我们接下来做一个具体测试：测试场景是在 getView 的时候通过 log 打印出 position 和当前系统时间。通过初始化 1000 条数据到 Adapter 显示在 ListView 中，然后滚动到底部，计算出 position=0 和 position=999 时的时间间隔。笔者用此场景对上述两种方案进行测试，测试结果如下。

(1) 优化建议测试结果

```
12-05 10:44:46.039: DEBUG/MyAdapter(13929): Position:0---1291517086043
12-05 10:44:46.069: DEBUG/MyAdapter(13929): Position:1---1291517086072
12-05 10:44:46.079: DEBUG/MyAdapter(13929): Position:2---1291517086085
...
12-05 10:45:04.109: DEBUG/MyAdapter(13929): Position:997---1291517104112
12-05 10:45:04.129: DEBUG/MyAdapter(13929): Position:998---1291517104135
12-05 10:45:04.149: DEBUG/MyAdapter(13929): Position:999---1291517104154
```

共耗时：17967

(2) 没优化的测试结果

```
12-05 10:51:42.569: DEBUG/MyAdapter(14131): Position:0---1291517502573
12-05 10:51:42.589: DEBUG/MyAdapter(14131): Position:1---1291517502590
12-05 10:51:42.609: DEBUG/MyAdapter(14131): Position:2---1291517502617
...
12-05 10:52:07.079: DEBUG/MyAdapter(14131): Position:998---1291517527082
12-05 10:52:07.099: DEBUG/MyAdapter(14131): Position:999---1291517527108
```

共耗时：24535

在 1000 条记录的情况下就有如此差距，一旦数据上万，或 ListView 的 Item 布局更加复杂的时候，优化的作用就更加突出了。

再考虑一个问题：假如在我们的列表中有 1000000 项时会发生什么情况呢？是不是会占用极大的系统资源？请读者看下面的代码：

```
public View getView(int position, View convertView, ViewGroup parent) {
View item = mInflater.inflate(R.layout.list_item_icon_text, null);
((TextView) item.findViewById(R.id.text)).setText(DATA[position]);
((ImageView) item.findViewById(R.id.icon)).setImageBitmap(
(position & 1) == 1 ? mIcon1 : mIcon2);
return item;
}
```

由此可见，如果超过 1000000 项时，后果不堪设想。再来看下面的演示代码：

```
public View getView(int position, View convertView, ViewGroup parent) {
if (convertView == null) {
convertView = mInflater.inflate(R.layout.item, null);
}
((TextView) convertView.findViewById(R.id.text)).setText(DATA[position]);
((ImageView) convertView.findViewById(R.id.icon)).setImageBitmap(
(position & 1) == 1 ? mIcon1 : mIcon2);
return convertView;
}
```

而上面的代码会好很多，系统将会减少创建很多 View 的情形，性能得到了很大的提升。究竟还有没有更优化的方法呢？再来看下面的演示代码。

```
public View getView(int position, View convertView, ViewGroup parent) {
ViewHolder holder;
if (convertView == null) {
convertView = mInflater.inflate(R.layout.list item icon text, null);
holder = new ViewHolder();
holder.text = (TextView) convertView.findViewById(R.id.text);
holder.icon = (ImageView) convertView.findViewById(R.id.icon);
convertView.setTag(holder);
} else {
holder = (ViewHolder) convertView.getTag();
}
holder.text.setText(DATA[position]);
holder.icon.setImageBitmap((position & 1) == 1 ? mIcon1 : mIcon2);
return convertView;
}
static class ViewHolder {
TextView text;
ImageView icon;
}
```

上述代码会不会给系统带来很大的提升呢？看看下面三种方式的性能对比，如图 7-1 所示。

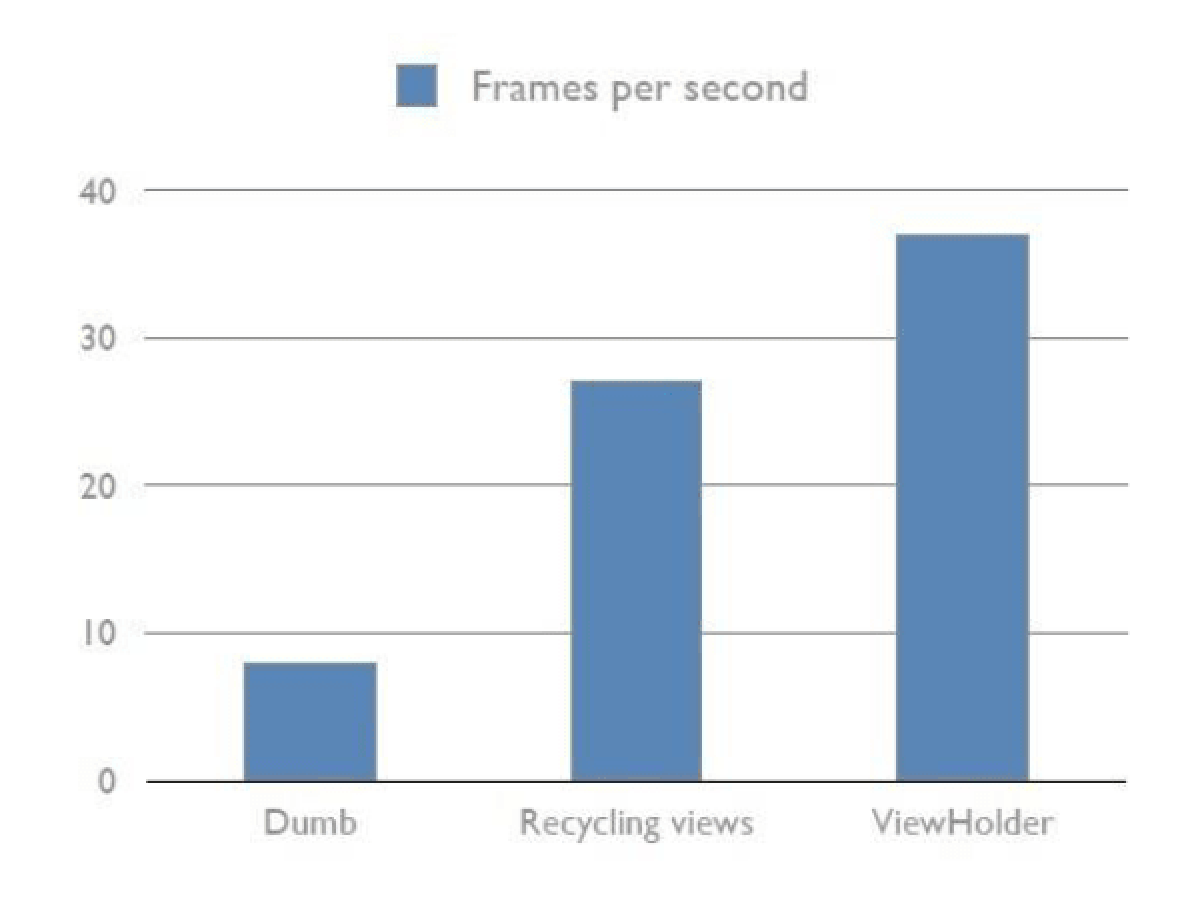


图 7-1　性能对比

事实正是如此，第三种性能更加高效。

7.4.3 ListView 异步加载图片优化

ListView 异步加载图片是非常实用的方法，只要通过网络获取图片资源，一般使用这种方法比较好，因为会有更好的用户体验。接下来通过一个具体实例的实现，来说明 ListView 异步加载图片优化的方法。

实例 1	
源码路径	\daima\7\AsyncListImage
功能	演示 ListView 异步加载图片优化

(1) 文件 AsyncImageLoader.java 实现了主方法，代码如下：

```
package cn.xxx.test;

import java.io.IOException;
import java.io.InputStream;
import java.lang.ref.SoftReference;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.HashMap;

import android.graphics.drawable.Drawable;
import android.os.Handler;
import android.os.Message;

public class AsyncImageLoader {

    private HashMap<String, SoftReference<Drawable>> imageCache;

        public AsyncImageLoader() {
         imageCache = new HashMap<String, SoftReference<Drawable>>();
        }

        public Drawable loadDrawable(final String imageUrl, final
ImageCallback imageCallback) {
            if (imageCache.containsKey(imageUrl)) {
                SoftReference<Drawable> softReference = imageCache.get(imageUrl);
                Drawable drawable = softReference.get();
                if (drawable != null) {
                    return drawable;
                }
            }
            final Handler handler = new Handler() {
                public void handleMessage(Message message) {
                   imageCallback.imageLoaded((Drawable) message.obj, imageUrl);
                }
            };
```



```
            new Thread() {
                @Override
                public void run() {
                    Drawable drawable = loadImageFromUrl(imageUrl);
                    imageCache.put(imageUrl, new SoftReference<Drawable>
(drawable));
                    Message message = handler.obtainMessage(0, drawable);
                    handler.sendMessage(message);
                }
            }.start();
            return null;
        }

        public static Drawable loadImageFromUrl(String url) {
            URL m;
            InputStream i = null;
            try {
                m = new URL(url);
                i = (InputStream) m.getContent();
            } catch (MalformedURLException e1) {
                e1.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
            Drawable d = Drawable.createFromStream(i, "src");
            return d;
        }

        public interface ImageCallback {
            public void imageLoaded(Drawable imageDrawable, String
imageUrl);
        }
}
```

上述代码是实现异步获取图片的主方法，SoftReference 是软引用，目的是更好的实现系统回收变量，重复的 URL 直接返回已有的资源，实现回调函数，让数据成功后，更新到 UI 线程。

(2) 接下来看如下两个辅助类文件，第一个辅助类文件 ImageAndText.java 的演示代码如下。

```
package cn.xxx.test;
public class ImageAndText {
        private String imageUrl;
        private String text;
        public ImageAndText(String imageUrl, String text) {
            this.imageUrl = imageUrl;
            this.text = text;
        }
        public String getImageUrl() {
            return imageUrl;
```

```
        }
        public String getText() {
            return text;
        }
}
```

第二个辅助类文件 ViewCache.java 的演示代码如下。

```
package cn.xxx.test;
import android.view.View;
import android.widget.ImageView;
import android.widget.TextView;
public class ViewCache {

        private View baseView;
        private TextView textView;
        private ImageView imageView;
        public ViewCache(View baseView) {
            this.baseView = baseView;
        }
        public TextView getTextView() {
            if (textView == null) {
                textView = (TextView) baseView.findViewById(R.id.text);
            }
            return textView;
        }
        public ImageView getImageView() {
            if (imageView == null) {
                imageView = (ImageView) baseView.findViewById(R.id.image);
            }
            return imageView;
        }
}
```

ViewCache 是辅助获取 adapter 的子元素布局。

(3) 再看实现 ListView 的 Adapter，对应文件 ImageAndTextListAdapter.java 的代码如下。

```
package cn.wangmeng.test;
import java.util.List;
import cn.wangmeng.test.AsyncImageLoader.ImageCallback;
import android.app.Activity;
import android.graphics.drawable.Drawable;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.TextView;
public class ImageAndTextListAdapter extends ArrayAdapter<ImageAndText> {
        private ListView listView;
```



```
        private AsyncImageLoader asyncImageLoader;
        public ImageAndTextListAdapter(Activity activity,
List<ImageAndText> imageAndTexts, ListView listView) {
            super(activity, 0, imageAndTexts);
            this.listView = listView;
            asyncImageLoader = new AsyncImageLoader();
        }
        public View getView(int position, View convertView, ViewGroup parent) {
            Activity activity = (Activity) getContext();
            // Inflate the views from XML
            View rowView = convertView;
            ViewCache viewCache;
            if (rowView == null) {
                LayoutInflater inflater = activity.getLayoutInflater();
               rowView = inflater.inflate(R.layout.image_and_text_row, null);
                viewCache = new ViewCache(rowView);
                rowView.setTag(viewCache);
            } else {
                viewCache = (ViewCache) rowView.getTag();
            }
            ImageAndText imageAndText = getItem(position);
            // Load the image and set it on the ImageView
            String imageUrl = imageAndText.getImageUrl();
            ImageView imageView = viewCache.getImageView();
            imageView.setTag(imageUrl);
            Drawable cachedImage = asyncImageLoader.loadDrawable(imageUrl,
new ImageCallback() {
                public void imageLoaded(Drawable imageDrawable, String imageUrl) {
                    ImageView imageViewByTag = (ImageView)
listView.findViewWithTag(imageUrl);
                    if (imageViewByTag != null) {
                        imageViewByTag.setImageDrawable(imageDrawable);
                    }
                }
            });
             if (cachedImage == null) {
                 imageView.setImageResource(R.drawable.default_image);
             }else{
                 imageView.setImageDrawable(cachedImage);
             }
            // Set the text on the TextView
            TextView textView = viewCache.getTextView();
            textView.setText(imageAndText.getText());
            return rowView;
        }
}
```

在上述代码中有一个技巧：imageView.setTag(imageUrl)，setTag 是存储数据的，这样是为了保证在回调函数时，listview 用于更新自己对应的 item。

(4) 最后看布局文件 main.xml，代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
            android:orientation="horizontal"
            android:layout width="fill parent"
            android:layout height="wrap content">
      <ImageView android:id="@+id/image"
                android:layout width="wrap content"
                android:layout height="wrap content"
                />
      <TextView android:id="@+id/text"
               android:layout width="wrap content"
               android:layout height="wrap content"/>
</LinearLayout>
```

这样 ListView 就通过异步加载的方式加载了指定的图片，实现了优化目的，提高了效率。执行效果如图 7-2 所示。



图 7-2　执行效果

7.5　优化 Android 图形

在 Android 应用中，有 2D 图形和 3D 图形两种，为了提高程序的运行效率，我们需要特意优化这些图形。在本节的内容中，将简要介绍优化 Android 图形的基本知识。

7.5.1　2D 绘图的基本优化

这里主要说的是 OpenGLes 用于 2D 时的优化，但是这些优化方法也适用于 3D 图形。基本优化原则如下。

(1) 谷歌提醒我们，不要在项目的交互过程中分配内存，尽量预先分配好，像写 C 一样写 Java。并且尽量少点函数调用，gl.gl**这类的函数会调到 jni，会比较耗时。

(2) 单线程和双线程的选择。

内置的 GLSurfaceView 会创建一个新的线程来更新场景及编制场景(同时还要处理主线程响应的一些事件)。很多开源的引擎用的是双线程(除主线程之外)，一个用来算场景及物理模型(物理线程)，一个用来绘制。一般手机只有一个 CPU，实际多线程是分时的，分多个线程还有线程之间的 context switch 上的消耗。这主要是因为 OpenGL 的绘制是异步



的，当调用 glDraw*及其他的 gles()函数，它们会比较快的返回，但真正的绘制发生在 eglSwapBuffer()时，所以在 eglSwapBuffers 时绘制线程会花很多时间等待 gles()的渲染结束。所以是用两个线程会快，而且用得好的话会快很多。

7.5.2 触发屏幕图形触摸器的优化

在 Android 模拟器中，当用鼠标单击一次模拟器屏幕然后释放，会先触发 ACTION_DOWN，然后 ACTION_UP。如果是在屏幕上移动那么才会触发 ACTION_MOVE 的动作。但是这只是模拟器的效果，接下来看一下真机与模拟器的区别。

当我们的用户在玩游戏的时候，尤其是 RPG 这种类型的游戏，肯定需要会长时间的去触屏按我们的虚拟按键，比如我们会在屏幕上画上一个虚拟方向盘类似这样子。那么其实 ACTION_MOVE 这个事件会被 Android 一直在响应。

为什么会一直响应 ACTION_MOVE 这个动作呢？如果用户没有移动手指而是静止不动也会一直响应？具体原因有如下两点：

- 第一点：因为 Android 对于触屏事件很敏感。
- 第二点：虽然我们的手指感觉是静止没有移动，其实事实不是如此！当我们的手指触摸到手机屏幕上之后，感觉静止没动，其实手指在不停地微颤。对此，读者可以具体尝试。

开始具体分析，如果 ACTION_MOVE 此时间一直被 Android 一直不停地响应并处理，无疑对我们游戏的性能增加了不少的负担。比如我们项目线程绘图时间每次用了 100ms，那么当手指触摸屏幕，这短暂的 0.1 秒内大概会产生 10 个左右的 MotionEvent，并且系统会尽可能快地把这些 event 发给监听线程，这样在这一段时间内 CPU 就会忙于处理 onTouchEvent，严重的话可能会造成画面一卡一卡的。

那么我们其实根本用不着按键响应这么多次，而只需在我们每次绘图后，或者绘图前接受一次用户按键即可，这样能让帧率不至于下降的太厉害。我们需要控制这个时间让它慢下来，随着我们的绘图时间一起来合作，这样就能减少系统线程的负担。

也可能有的读者会问为什么不用 sleep()的方法，其实如果只是想让线程休眠指定时间的话可以用 sleep()函数，但是这个没有资源锁的限制。而 Object 的 wait()和 notify()方法通常用在时间不定的条件限制等待，并且必须写在同步代码块中。

也可能有的读者会问为什么不用当前类的 object 来使用：this.wait()，而是 new 一个 object。因为 synchronized 中的 Object 表示 Object 调用 wait()必须拥有该对象的监视锁，当前我们有了 object 的锁，就要用 object 调用 wait()。

注意： 读者要慎用方法 Object.wait(long timeout)，因为在测试时发现这个睡眠时间其实比我们规定的时间要略微长一些，不过如果合理控制好时间还是没问题的。

7.5.3 SurfaceView 绘图覆盖刷新及脏矩形刷新方法

脏矩形是指每次都重绘整个背景图，其实这是非常浪费的，前后两帧的图其实只有很少的一部分发生了变化，因此可以只重绘变化的部分。这是一种常用的绘图优化方式，需要注意的是，Android 用了双缓冲，也就是说当使用脏矩形的时候，需要连续绘制两次才能

完成对 surface 的刷新。

SurfaceView 在 Android 中用作游戏开发是最适宜的，本文就将演示游戏开发中常用的两种绘图刷新策略在 SurfaceView 中的实现方法。

首先我们来看一下本例需要用到的两个素材图片，分别如图 7-3 和图 7-4 所示。



图 7-3　渐变图

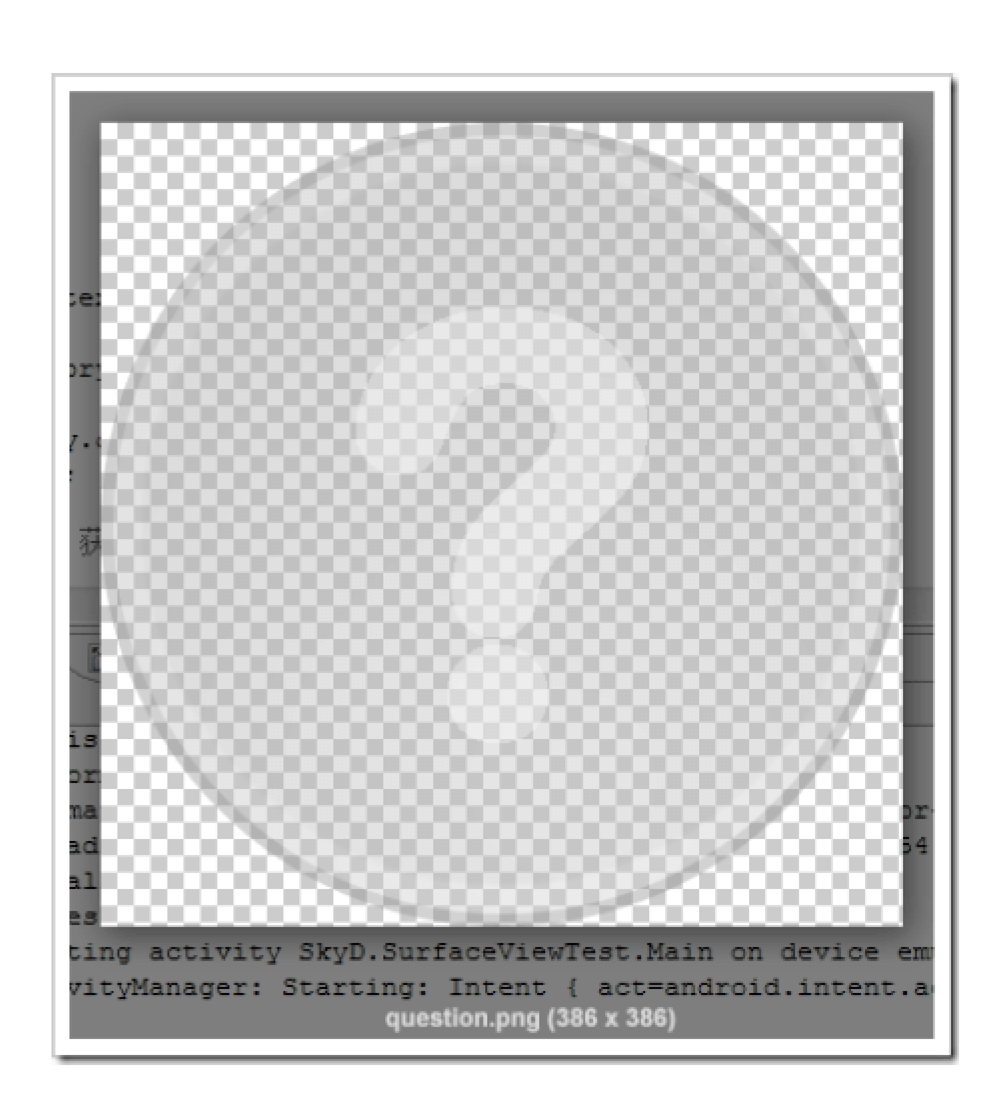


图 7-4　半透明的图像

图 7-3 作为背景图片，图 7-4 是一个半透明的图像，我们希望将它放在上面，围绕其圆心不断旋转。请读者先看如下代码：

```
package SkyD.SurfaceViewTest;
import android.app.Activity;
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.os.Bundle;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
```



```
public class Main extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(new MySurfaceView(this));
    }
    // 自定义的 SurfaceView 子类
    class MySurfaceView extends SurfaceView implements
SurfaceHolder.Callback {
       // 背景图
       private Bitmap BackgroundImage;
       // 问号图
       private Bitmap QuestionImage;
       SurfaceHolder Holder;
       public MySurfaceView(Context context) {
          super(context);
          BackgroundImage = BitmapFactory.decodeResource(getResources(),
                R.drawable.bg);
          QuestionImage = BitmapFactory.decodeResource(getResources(),
                R.drawable.question);
          Holder = this.getHolder();// 获取 holder
          Holder.addCallback(this);
       }
       @Override
       public void surfaceChanged(SurfaceHolder holder, int format, int width,
             int height) {
          // TODO Auto-generated method stub
       }
       @Override
       public void surfaceCreated(SurfaceHolder holder) {
          // 启动自定义线程
          new Thread(new MyThread()).start();
       }
       @Override
       public void surfaceDestroyed(SurfaceHolder holder) {
          // TODO Auto-generated method stub
       }
       // 自定义线程类
       class MyThread implements Runnable {
          @Override
          public void run() {
             Canvas canvas = null;
             int rotate = 0;// 旋转角度变量
             while (true) {
                try {
                   canvas = Holder.lockCanvas();// 获取画布
                   Paint mPaint = new Paint();
                   // 绘制背景
                   canvas.drawBitmap(BackgroundImage, 0, 0, mPaint);
                   // 创建矩阵以控制图片旋转和平移
                   Matrix m = new Matrix();
```

```
                    // 设置旋转角度
                    m.postRotate((rotate += 48) % 360,
                          QuestionImage.getWidth() / 2,
                          QuestionImage.getHeight() / 2);
                    // 设置左边距和上边距
                    m.postTranslate(47, 47);
                    // 绘制问号图
                    canvas.drawBitmap(QuestionImage, m, mPaint);
                    // 休眠以控制最大帧频为每秒约 30 帧
                    Thread.sleep(33);
                } catch (Exception e) {
                } finally {
                    Holder.unlockCanvasAndPost(canvas);// 解锁画布，提交画好的图像
                }
            }
        }
      }
   }
}
```

执行后的效果如图 7-5 所示。



图 7-5　执行效果

上述代码的运行效果符合我们的要求，但是有一个问题：在代码中设置的帧频最大值是每秒 30 帧，而实际运行时的帧频根据目测就能看出是到不了 30 帧的，这是因为程序在每一帧都要对整个画面进行重绘，过多的时间都被用作绘图处理，所以难以达到最大帧频。

接下来我们将采取脏矩形刷新的方法来优化性能，所谓脏矩形刷新，意为仅刷新有新变化的部分所在的矩形区域，而其他没用的部分就不去刷新，以此来减少资源浪费。我们可以通过在获取 Canvas 画布时，为其指派一个参数来声明我们需要画布哪个局部，这样就可以只获得这个部分的控制权。例如下面的演示代码：

```
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        // TODO Auto-generated method stub

    }

    // 自定义线程类
    class MyThread implements Runnable {

        @Override
        public void run() {
            Canvas canvas = null;
            int rotate = 0;// 旋转角度变量
            int frameCount = 0;
            while (true) {
                try {
                    canvas = Holder.lockCanvas(new Rect(47,47,240,240));// 获取画布
                    Paint mPaint = new Paint();
                    if (frameCount++ < 1) {
                        // 绘制背景
                        canvas.drawBitmap(BackgroundImage, 0, 0, mPaint);
                    }
                    // 创建矩阵以控制图片旋转和平移
                    Matrix m = new Matrix();
                    // 设置旋转角度
                    m.postRotate((rotate += 48) % 360,
                            QuestionImage.getWidth() / 2,
                            QuestionImage.getHeight() / 2);
                    // 设置左边距和上边距
                    m.postTranslate(47, 47);
                    // 绘制问号图
                    canvas.drawBitmap(QuestionImage, m, mPaint);
                    // 休眠以控制最大帧频为每秒约30帧
                    Thread.sleep(33);
                } catch (Exception e) {
                } finally {
                    Holder.unlockCanvasAndPost(canvas);// 解锁画布，提交画好的图像
                }
```

如果这样改后直接运行的话，会发现问号图案会变得有残影了，如图 7-6 所示。

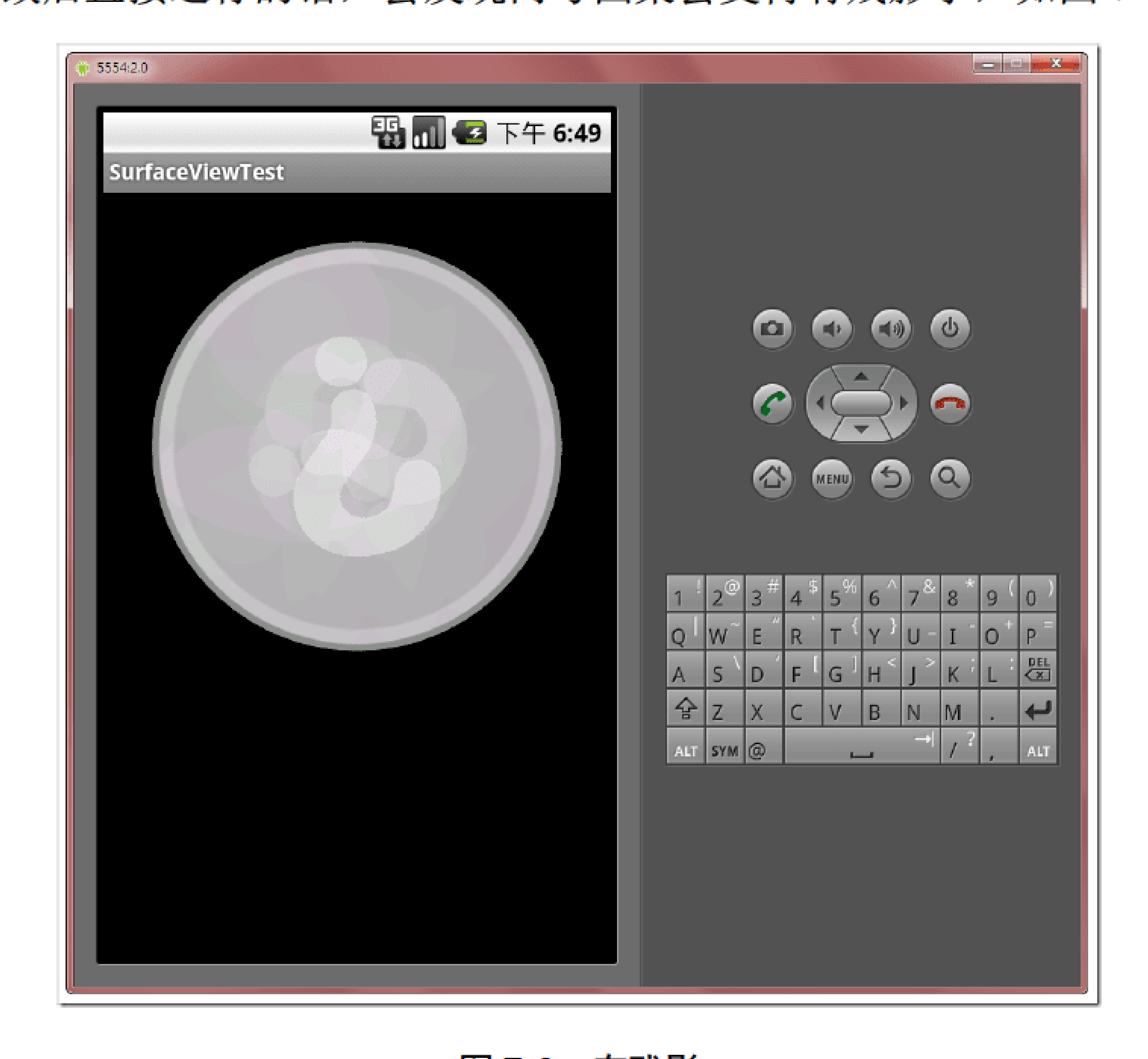


图 7-6　有残影

这正是我们使用半透明图案作范例的目的，通过这个重影可以看出，覆盖刷新其实就是将每次的新图形绘制到上一帧去，所以如果图像是半透明的，就要考虑重复叠加导致的问题了，而如果是完全不透明的图形则不会有任何问题。

再看背景会在背景图和黑色背景之间来回闪的问题，这个问题其实是源于SurfaceView的双缓冲机制，也就是说它会缓冲前两帧的图像交替传递给后面的帧用作覆盖，这样由于仅在第一帧绘制了背景，第二帧就是无背景状态了，且通过双缓冲机制一直保持下来，解决办法就是改为在前两帧都进行背景绘制：

```
if (frameCount++ < 2) {
    // 绘制背景
    canvas.drawBitmap(BackgroundImage, 0, 0, mPaint);
}
```

此时执行后就没有问题了，如图7-7所示。

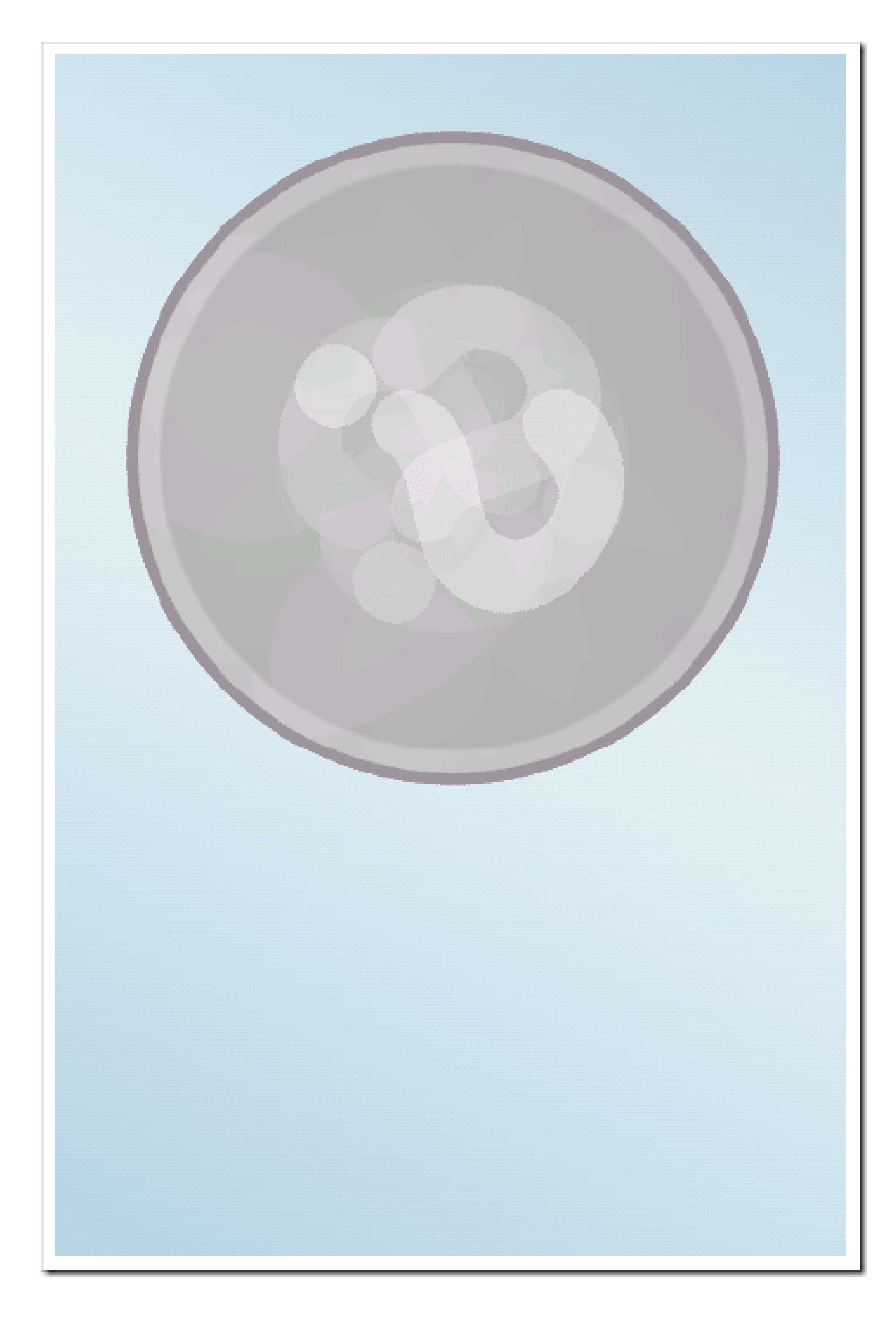


图7-7　执行效果

虽然此时还是达不到最大帧频，但是在真机上跑得会更快些，已经接近最大帧频了。

第8章 性能优化

在科学术语中，性能是指测量仪器仪表实现预期功能的能力的特性。在 Android 应用中，各个软件和内在系统的能力特性就是性能，例如资源存储、加载能力、虚拟机和渲染机制等。在本章的内容中，将详细讲解 Android 性能优化的基本知识，希望读者专心学习本章内容，为步入本书后面高级知识的学习打下基础。

8.1　资源存储优化

资源存储是指保存项目需要的资源，例如项目需要的图片素材、音频素材、布局文件和值文件等。本节将详细讲解资源存储优化的基本知识。

8.1.1　Android 文件存储

Android 的文件存储包括内部存储、外部存储和资源文件三种。

1. 内部存储

Android 允许应用程序创建仅能够自身访问的私有文件，通常保存在内部存储器上的如下目录：

```
/data/data/<package name>/files
```

内部存储支持标准 Java 的 IO 类，也提供了简化读写流式文件过程的函数，主要有如下两个函数：

- openFileOutput()
- openFileInput()

其中函数 openFileOutput()的功能是，为写入数据做准备而打开应用程序私有文件，若不存在则创建一个。例如下面的演示代码：

```
public FileOutputStream openFileOutput(String name,int mode);
```

其中第一个参数表示文件名，第二个参数表示操作模式。操作模式有如下 4 种：

- MODE_PRIVATE：私有模式；
- MODE_APPEND：追加模式；
- MODE_WORLD_READABLE：全局读；
- MODE_WORLD_WRITEABLE：全局写。

例如下面的演示代码：

```
String FILE_NAME="fileDemo.txt";                          //定义文件名
FileOutputStream fos=openFileOutput(FILE_NAME, Context.MODE_PRIVATE); //
以私有模式创建文件
String text="Some data";
fos.write(text.getBytes());                          //写入数据
fos.flush();                                         //将缓冲区剩余数据写入文件
fos.close();                                         //关闭 FileOutputStream
```

而函数 openFileInput()的功能是为读取数据做准备，打开应用程序的私有文件。例如下面的演示代码：

```
String FILE_NAME="fileDemo.txt";
FileInputStream fis=openFileInput(FILE_NAME);
```



```
byte[] readBytes=new byte[fis.available()];
while(fis.read(resdBytes)!=-1){
}
```

在具体应用时应该包含在“try/catch”块内。

2. 外部存储

外部存储是指 SD 卡使用的是 FAT 文件系统，可以通过 Linux 文件系统的文件访问权限的控制保证私密性。Android 模拟器不带 SD 卡，需要手动添加映像。使用<Android SDK>/tools 目录下的 mksdcard 工具可以创建映像文件，格式是：

```
mksdcard -l SDCARD E:\android\sdcard_file
```

其中参数分别代表 SD 卡标签、容量和保存位置。

如果想让模拟器启动时会自动加载 SD 卡，需要在模拟器的 Run Configurations 里设置，指明具体的 SD 卡路径即可。格式是：

```
-sdcard E:\Android\sdcard_file
```

在编程时需要检测/sdcard 目录是否可用，之后便可以使用标准 Java IO 实现文件操作。例如下面的演示代码：

```
String fileName = "SdcardFile-"+System.currentTimeMillis()+".txt"; //保证文件名不同
    File dir = new File("/sdcard/");
    if (dir.exists() && dir.canWrite()) { //检查目录存在性
        File newFile = new File(dir.getAbsolutePath() + "/" + fileName);
        FileOutputStream fos  = null;
        try {
            newFile.createNewFile();
            if (newFile.exists() && newFile.canWrite()) { //文件存在性检查
            ...
            }
        }
    }
```

3. 读取 XML 格式文件

在读取 XML 格式文件时，通过资源对象函数 getXml()获取解析器 XmlPullParser，例如下面的演示代码：

```
parser.next() != XmlPullParser.END_DOCUMENT //获取解析事件进行对比
```

XML 的事件类型如下：

- START_TAG：读取到游标开始标志；
- TEXT：读取到文本内容；
- END_TAG：读取到游标结束标志；
- END_DOCUMENT：文档末尾；
- getName()：获取元素名称；
- getAttributeCount()：获取元素的属性数量；

- getAttributeName()：获取属性名称；
- getAttributeValue()：获取值。

注意：读取 Android 资源文件的知识，在 8.1.2 节中进行专门讲解。

8.1.2　Android 中的资源存储

在 Android 开发中，我们离不开资源文件的使用，从 drawable 到 string，再到 layout，这些资源都为我们的开发提供了极大的便利，不过我们平时大部分时间接触的资源目录一般都是如下三个。

- /res/drawable：保存素材文件，例如图片；
- /res/values：保存值文件，通常命名为 strings.xml；
- /res/layout：保存布局文件，通常命名为 main.xml。

上述三个文件在 Android 项目目录中的结果一般如图 8-1 所示。

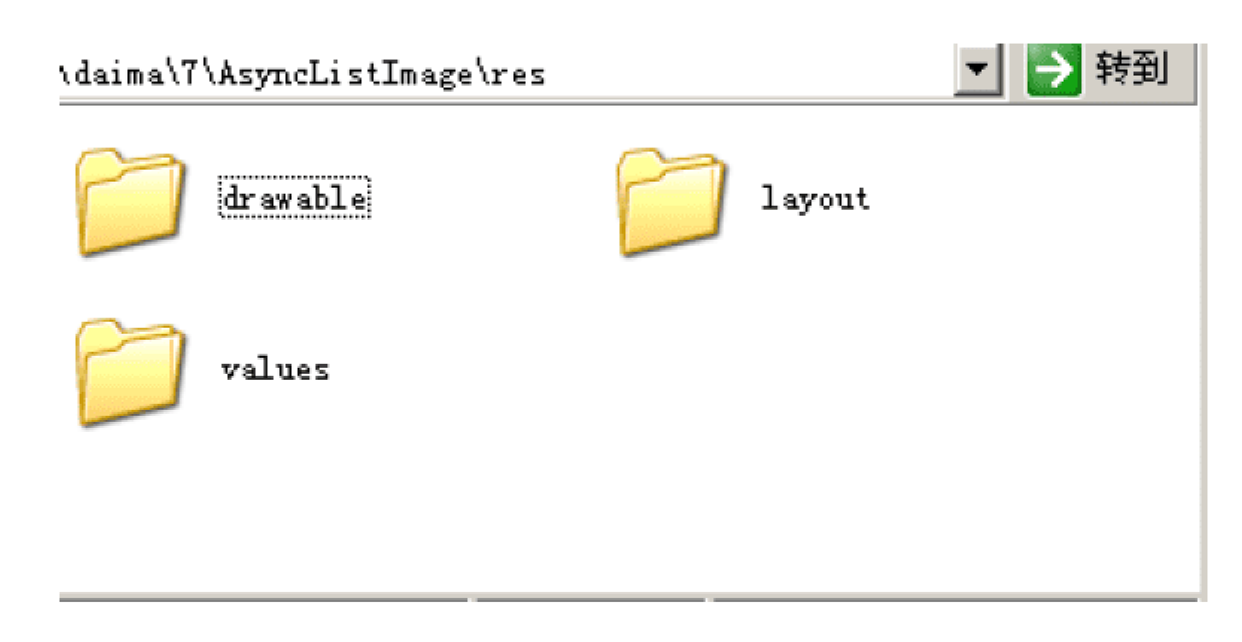


图 8-1　Android 项目中的资源存储目录

其实 Android 的资源文件并不止这些，接下来为大家介绍如下另外三个资源目录。

- /res/xml
- /res/raw
- /assets

(1) /res/xml 目录

首先是/res/xml，大家可能偶尔用到过这个目录，可以用来存储“.xml”格式的文件，并且和其他资源文件一样，这里的资源是会被编译成二进制格式放到最终的安装包里的。也可以通过 R 类来访问此文件，并且解析里面的内容，例如存放了一个名为 data.xml 的文件：

```
<?xml version="1.0" encoding="utf-8"?>
<root>
      <title>Hello XML!</title>
</root>
```

然后就可以通过资源 ID 来访问并解析这个文件了，例如下面的代码：

```
XmlResourceParser xml = getResources().getXml(R.xml.data);
xml.next();
int eventType = xml.getEventType();
boolean inTitle = false;
while(eventType != XmlPullParser.END_DOCUMENT) {
```

```
        //到达 title 节点时标记一下
        if(eventType == XmlPullParser.START TAG) {
                if(xml.getName().equals("title")) {
                        inTitle = true;
                }
        }

        //如过到达标记的节点则取出内容
        if(eventType == XmlPullParser.TEXT && inTitle) {
                ((TextView)findViewById(R.id.txXml)).setText(
                                xml.getText()
                );
        }
        xml.next();
        eventType = xml.getEventType();
}
```

在上述代码中，使用了资源类的方法 getXml()，返回了一个 XML 解析器，这个解析器的工作原理和 SAX 方式差不多。在此要注意的是，这里的 XML 文件最终会被编译成二进制形式。如果想让文件原样存储的话，那么就要用到下一个目录：/res/raw。

(2) /res/raw 目录

这个目录和/res/raw 目录的唯一区别是：里面的文件会原封不动地存储到设备上，不会被编译为二进制形式，访问的方式也是通过 R 类，例如下面的演示代码：

```
            ((TextView)findViewById(R.id.txRaw)).setText(
                    readStream(getResources().openRawResource(R.raw.rawtext))
            );

            private String readStream(InputStream is) {

                    try {
                            ByteArrayOutputStream bo = new
ByteArrayOutputStream();
                            int i = is.read();
                            while(i != -1) {
                                    bo.write(i);
                                    i = is.read();
                            }

                            return bo.toString();
                    } catch (IOException e) {
                            return "";
                    }
            }
```

在此使用了资源类中的方法 openRawResource()，返回给我们一个输入流，这样就可以任意读取文件中的内容了。例如像上述代码中那样，会原样输出文本文件中的内容。当然，如果需要更高的自由度，尽量不受 Android 平台的约束，那么/assets 目录就是首选了。

(3) /assets 目录

在此目录中的文件，除了不会被编译成二进制形式之外，并且访问方式是通过文件名，而不是资源 ID。并且还有更重要的一点就是，在此可任意建立子目录，而“/res”目录中的资源文件是不能自行建立子目录的。如果需要这种灵活的资源存储方式，请读者再看下面的演示代码：

```
AssetManager assets = getAssets();
 ((TextView)findViewById(R.id.txAssets)).setText(
                readStream(assets.open("data.txt"))
);
```

在 context 的上下文中，调用 getAssets()返回一个 AssetManager，然后使用 open()方法就可以访问需要的资源了，这里 open()方法是以 assets 目录为根的。所以上面这段代码访问的是 assets 目录中名为 data.txt 的资源文件。

8.1.3 Android 资源的类型和命名

(1) 资源文件的种类

从资源文件的类型来划分，我们可以将资源文件划分成 xml、图像和其他。以 xml 文件形式存储的资源可以放在 res 目录中的不同目录里，用来表示不同种类的资源，而图像资源会放在 res\drawable 目录中，除此之外，可以将任意的资源嵌入到 android 应用程序中，比如音频和视频等，一般这些资源放在 res\raw 目录中。

Android 支持的资源类型如下：

- res\values：类型是 xml，用于保存字符串、颜色、尺寸、类型、主题等资源，可以是任意文件名，对于字符串、颜色、尺寸等信息采用 key-value 形式表示，对于类型、主题等资源，采用其他形式表示。
- res\layout：类型是 xml，用于保存布局信息，一个资源文件表示一个 view 或 viewGroup 的布局。
- res\menu：类型是 xml，用于保存菜单资源。一个资源文件表示一个菜单(含子菜单)。
- res\anim：类型是 xml，用于保存与动画相关的信息，可以定义帧(frame)动画和补间(tween)动画。
- res\xml：类型是 xml，在此目录中的文件可以是任意类型的 xml 文件，这些 xml 文件可以在运行时被读取。
- res\raw：类型是任意类型，在该目录中的文件虽然也会被封装在 apk 文件中，但不会被编译，可以放置任意类型的文件，例如，各种类型的文档、音频、视频文件等。
- res\drawable：类型是图像，该目录中文件可以是多种格式的图像文件，例如，bmp、png、gif、jpg 等。图像不需要分辨率非常高，aapt 工具会优化这个目录中的图像文件。如果想按照字流读取该目录下的图像文件，需要将图像文件放在 res\raw 目录中。
- assets：是任意类型，该目录中的资源与 res\raw 中的资源一样，也不会被编译，

但不同的是该目录中的资源文件都不会生出资源 ID。

(2) 资源文件的命名

每个资源文件或者资源文件中的 key-value 对都会在 ADT 自动生成的 R 类(在 R.java 文件中)中找到相对应的 ID，其中资源文件名或 key-value 对中的 key 就是 R 类中的 java 变量名，因此，资源文件名 key 的命名首先要符合 java 变量的命名规则。

除了资源文件和 key 本身的命名要遵循相应的规则外，多个资源文件和 key 也要遵循唯一的原则，也就是说，同类资源的文件名或 key 不能重复，就算这两个 key 在不同的 xml 文件中也不行。

由于 ADT 在生成 ID 时并不考虑资源文件的扩展名，因此，在 res\drawable、res\raw 等目录中不能存在文件名相同、扩展名不同的资源文件。例如，在 res\drawable 目录中不能同时放置 icon.jpg 和 icon.png 文件。

8.1.4　Android 文件资源(raw/data/asset)的存取

对于其他几个目录，读者应该十分熟悉。本节将着重讲解 raw/data/asset 目录的知识，介绍存取 raw/data/asset 资源的方法。

(1) 私有文件夹下的文件存取(/data/data/包名)

在私有文件夹中存取/data/data/资源的演示代码如下：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import org.apache.http.util.EncodingUtils;
public void writeFileData(String fileName,String message){
    try{
     FileOutputStream fout = openFileOutput(fileName, MODE PRIVATE);
     byte [] bytes = message.getBytes();
     fout.write(bytes);
      fout.close();
     }
    catch(Exception e){
     e.printStackTrace();
    }
}
public String readFileData(String fileName){
     String res="";
     try{
      FileInputStream fin = openFileInput(fileName);
      int length = fin.available();
      byte [] buffer = new byte[length];
      fin.read(buffer);
      res = EncodingUtils.getString(buffer, "UTF-8");
      fin.close();
     }
     catch(Exception e){
      e.printStackTrace();
     }
```

```
        return res;
      }
```

(2) 从 raw 读取资源

从 resource 中的 raw 文件夹中获取文件并读取数据，这里的资源文件只能读不能写。演示代码如下：

```
    public String getFromRaw(String fileName){
        String res = "";
          try{
           InputStream in = getResources().openRawResource(R.raw.test1);
           int length = in.available();
           byte [] buffer = new byte[length];
           in.read(buffer);
           res = EncodingUtils.getString(buffer, "UTF-8");
           in.close();
          }
          catch(Exception e){
           e.printStackTrace();
          }
        return res ;
      }
```

(3) 从 asset 读取资源

接下来讲解从 asset 中获取文件并读取数据的方法，这里的资源文件只能读不能写。演示代码如下：

```
    public String getFromAsset(String fileName){
        String res="";
        try{
         InputStream in = getResources().getAssets().open(fileName);
         int length = in.available();
           byte [] buffer = new byte[length];
           in.read(buffer);
           res = EncodingUtils.getString(buffer, "UTF-8");
        }
        catch(Exception e){
         e.printStackTrace();
        }
        return res;
      }
```

8.1.5 Android 对 Drawable 对象的优化

Android 中的 Drawable 对编写程序是非常有用的。Drawable 通常是一个与 view 相关的画图容器。例如一个 aBitmapDrawable 是用来显示图片的，一个 ShapeDrawable 是用来画图和渐变的，甚至可以通过它创建负责的渲染。

Drawables 允许我们不需要继承就可以很容易地定制 widgets 渲染。事实是，Android 的应用程序和 widgets 是使用该 Drawable 对象的，在 Android 的核心框架中大约有 700 个



Drawable 被使用。正是因为它是如此广泛地被使用，所以 Android 对它进行了优化。例如，当每一次创建一个按钮时，一个新的 Drawable 就会被装载，这就意味着应用程序中所有使用不同 Drawable 对象实现不同背景的按钮，所有的 Drawable 对象共用一个公用的状态，我们称这个状态为“constant state(常态)”。在这个状态内，根据我们使用的不同 Drawable 对象而不同，但是它通常包括一个资源所有的属性。以按钮为例，常态包括一个位图。如此一来所有按钮就可以共享一张位图，这将会节省很多资源。

图 8-2 介绍了设置一张图给两个不同 View 作为背景的创建过程。正如我们所看到的那样，创建两个 Drawable 后，共享公共的部分是同一张位图。

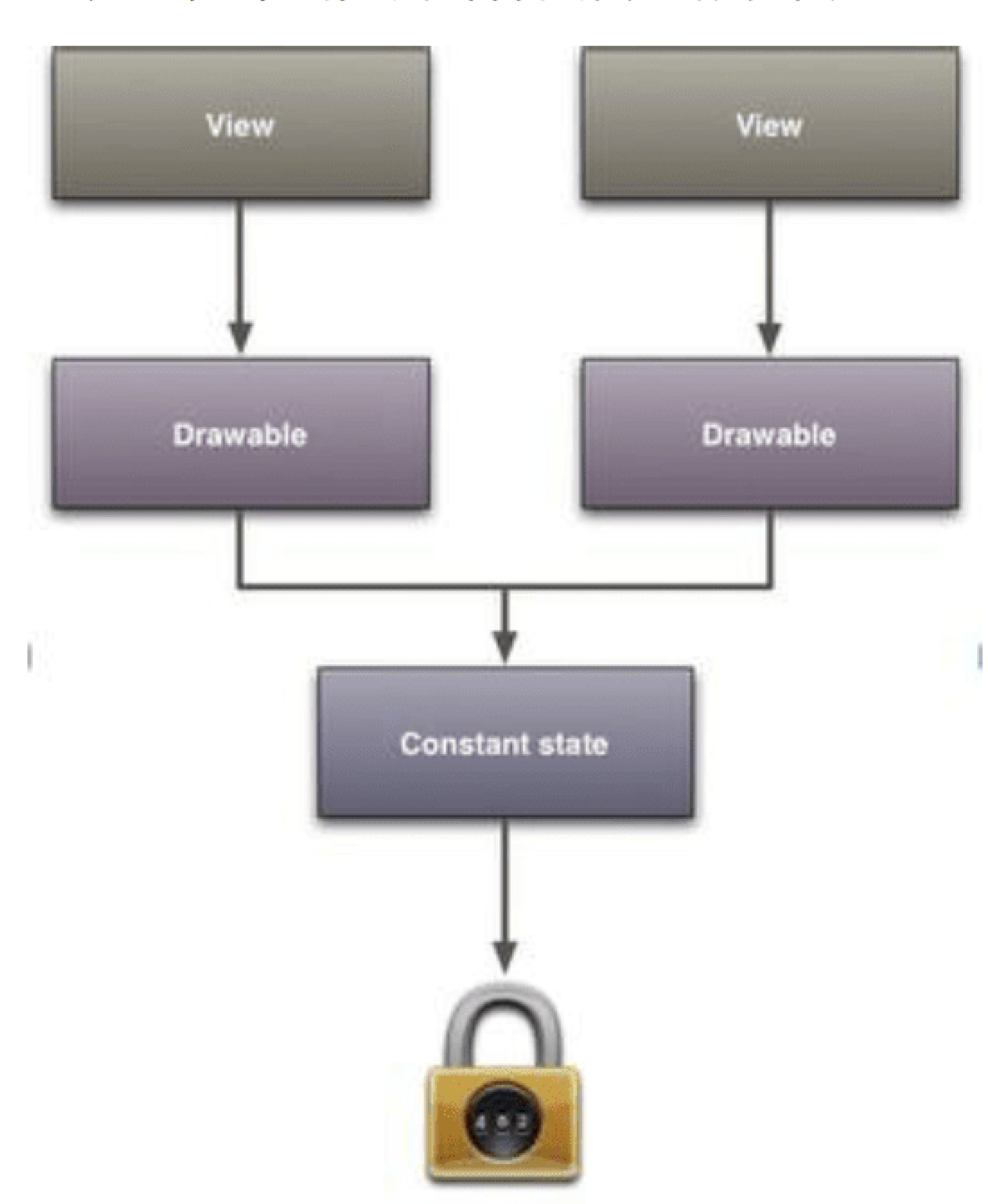


图 8-2　一张图给两个不同 View 作为背景

上述“状态分享”特点极大地避免了内存浪费，但是当我们试图去修改 Drawable 的属性时会导致一些问题。假设是关于书的列表程序，书名之后，当你标注为喜欢的时候显示为不透明，而标注不喜欢的时候显示为完全透明的星星。为了达到这样的效果，也许会在我们的 adapter 的 getView 实现下面的方法：

```
Book book = ...;
TextView listItem = ...;
listItem.setText(book.getTitle());
Drawable star = context.getResources().getDrawable(R.drawable.star);
if (book.isFavorite()) {
  star.setAlpha(255); // opaque
} else {
  star.setAlpha(70); // translucent
}
```

但是不幸的是，上述代码会有一个很奇怪的结果，所有的 Drawable 对象都会有相同的

透明值。这种结果可以“constant state(常态)”来解释，因为当我们从一个 list item 中获取一个 drawable 对象时，“constant state”是一样的，对 BitmapDrawable 来说，透明值就是一个常态，因此对于改变一个 Drawable 对象实例的透明值来说，会改变所有其他对象的透明值。在 Android 1.5 或者更好的设备上，可以通过方法 mutate()很容易地解决这个问题。当我们对一个 Drawable 对象调用这个方法时，Drawable 对象会被复制而不会影响其他对象。在此记住 Bitmap 对象依旧是被重用的，即使使用的是 mutate()。图 8-3 说明了调用 mutate()对象之后的情况。

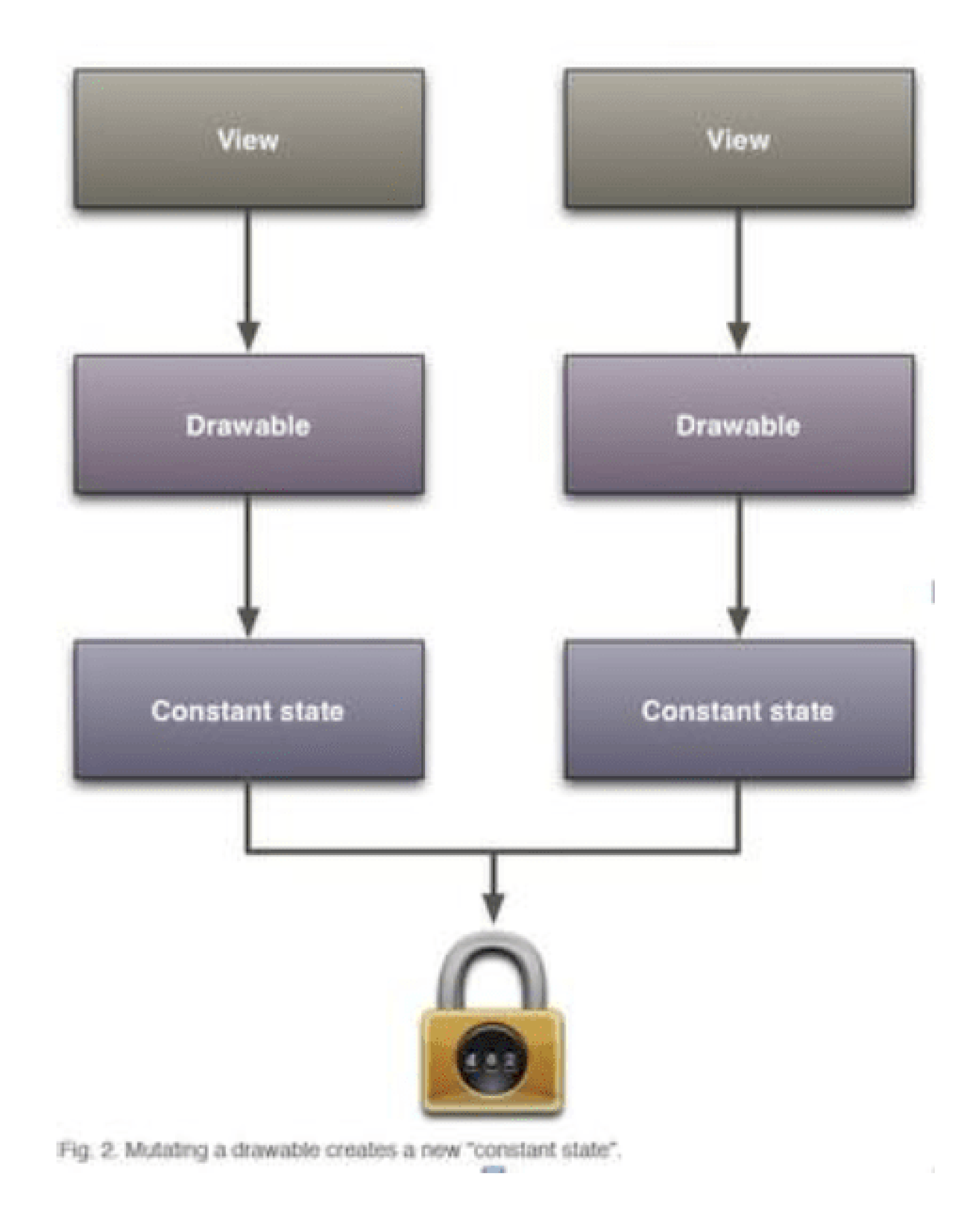


图 8-3　调用 mutate()对象后

接下来更新一下我们的代码：

```
<div>
<p align="left">Drawable star =
context.getResources().getDrawable(R.drawable.star);<br>
if (book.isFavorite()) {<br>
  star.mutate().setAlpha(255); // opaque<br>
} else {<br>
  star. mutate().setAlpha(70); // translucent<br>
}</p>
</div>
```

为了方便方法 mutate()返回的是 Drawable 对象自己，这就允许我们采用链的方法调用，它不会产生新的对象，通过上面的代码片段，程序行为会变得正常。

8.1.6　建议使用 Drawable，而不是 Bitmap

在 Android 应用中，Drawable 和 Bitmap 都能实现图像加载机制。但是遵循性能优化的原则，建议使用 Drawable。看我们下面的测试。

(1) 首先通过如下代码测试加载 1000 个 Drawable 对象。

```
public class Main extends Activity {
    int number = 1000;
    Drawable[] array;
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
        array = new BitmapDrawable[number];
        for(int i = 0; i < number; i++)
        {
            Log.e("", "测试第" + (i+1) + "张图片");
            array[i] = getResources().getDrawable(R.drawable.img);
        }
    }
}
```

输出结果是：

```
04-12 21:49:25.248: D/szipinf(7828): Initializing inflate state
04-12 21:49:25.398: E/(7828): 测试第 1 张图片
04-12 21:49:25.658: D/dalvikvm(7828): GC EXTERNAL ALLOC freed 48K, 50%
free 2692K/5379K, external 0K/0K, paused 24ms
04-12 21:49:25.748: E/(7828): 测试第 2 张图片
04-12 21:49:25.748: E/(7828): 测试第 3 张图片
……
……
04-07 21:49:26.089: E/(7828): 测试第 998 张图片
04-07 21:49:26.089: E/(7828): 测试第 999 张图片
04-07 21:49:26.089: E/(7828): 测试第 1000 张图片
```

由此可见，程序能够正常运行，加载 1000 个 Drawable 对象完全没有没问题。

(2) 然后通过如下代码测试加载 1000 个 Bitmap 对象。

```
public class Main extends Activity {
    int number = 1000;
    Bitmap bitmap[];
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
       bitmap = new Bitmap[number];
       for (int i = 0; i < number; i++)
       {
           Log.e("", "测试第" + (i+1) + "张图片");
           bitmap[i] = BitmapFactory.decodeResource(getResources(),
R.drawable.img);
       }
    }
}
```

输出结果是：

```
04-12 22:06:05.344: D/szipinf(7937): Initializing inflate state
04-12 22:06:05.374: E/(7937): 测试第 1 张图片
04-12 22:06:05.544: D/dalvikvm(7937): GC EXTERNAL ALLOC freed 51K, 50%
free 2692K/5379K, external 0K/0K, paused 40ms
04-12 22:06:05.664: E/(7937): 测试第 2 张图片
04-12 22:06:05.774: D/dalvikvm(7937): GC EXTERNAL ALLOC freed 1K, 50%
free 2691K/5379K, external 6026K/7525K, paused 31ms
04-12 22:06:05.834: E/(7937): 测试第 3 张图片
04-12 22:06:05.934: D/dalvikvm(7937): GC EXTERNAL ALLOC freed <1K, 50%
free 2691K/5379K, external 12052K/14100K, paused 24ms
04-12 22:06:06.004: E/(7937): 测试第 4 张图片
04-12 22:06:06.124: D/dalvikvm(7937): GC EXTERNAL ALLOC freed <1K, 50%
free 2691K/5379K, external 18128K/20126K, paused 27ms
04-12 22:06:06.204: E/(7937): 测试第 5 张图片
04-12 22:06:06.315: D/dalvikvm(7937): GC EXTERNAL ALLOC freed <1K, 50%
free 2691K/5379K, external 24104K/26152K, paused 26ms
04-12 22:06:06.395: E/(7937): 测试第 6 张图片
04-12 22:06:06.495: D/dalvikvm(7937): GC EXTERNAL ALLOC freed <1K, 50%
free 2691K/5379K, external 30130K/32178K, paused 22ms
04-12 22:06:06.565: E/(7937): 测试第 7 张图片
04-12 22:06:06.665: D/dalvikvm(7937): GC EXTERNAL ALLOC freed <1K, 50%
free 2691K/5379K, external 36156K/38204K, paused 22ms
04-12 22:06:06.745: E/(7937): 测试第 8 张图片
04-12 22:06:06.845: D/dalvikvm(7937): GC EXTERNAL ALLOC freed 2K, 51%
free 2689K/5379K, external 42182K/44230K, paused 23ms
04-12 22:06:06.845: E/dalvikvm-heap(7937): 6171224-byte external
allocation too large for this process.
04-12 22:06:06.885: I/dalvikvm-heap(7937): Clamp target GC heap from
48.239MB to 48.000MB
04-12 22:06:06.885: E/GraphicsJNI(7937): VM won't let us allocate
6171224 bytes
04-12 22:06:06.885: D/dalvikvm(7937): GC FOR MALLOC freed <1K, 51% free
2689K/5379K, external 42182K/44230K, paused 25ms
04-12 22:06:06.885: D/AndroidRuntime(7937): Shutting down VM
04-12 22:06:06.885: W/dalvikvm(7937): threadid=1: thread exiting with
uncaught exception (group=0x40015560)
04-12 22:06:06.885: E/AndroidRuntime(7937): FATAL EXCEPTION: main
04-12 22:06:06.885: E/AndroidRuntime(7937): java.lang.OutOfMemoryError:
bitmap size exceeds VM budget
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.graphics.Bitmap.nativeCreate(Native Method)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.graphics.Bitmap.createBitmap(Bitmap.java:477)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.graphics.Bitmap.createBitmap(Bitmap.java:444)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.graphics.Bitmap.createScaledBitmap(Bitmap.java:349)
```

```
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.graphics.BitmapFactory.finishDecode(BitmapFactory.java:498)

04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.graphics.BitmapFactory.decodeStream(BitmapFactory.java:473)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.graphics.BitmapFactory.decodeResourceStream(BitmapFactory.java:336)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.graphics.BitmapFactory.decodeResource(BitmapFactory.java:359)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.graphics.BitmapFactory.decodeResource(BitmapFactory.java:385)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
bassy.test.drawable.Main.onCreate(Main.java:37)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1047)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.app.ActivityThread.performLaunchActivity(ActivityThread.java:1722)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:1784)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.app.ActivityThread.access$1500(ActivityThread.java:123)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.app.ActivityThread$H.handleMessage(ActivityThread.java:939)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.os.Handler.dispatchMessage(Handler.java:99)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.os.Looper.loop(Looper.java:130)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
android.app.ActivityThread.main(ActivityThread.java:3835)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
java.lang.reflect.Method.invokeNative(Native Method)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
java.lang.reflect.Method.invoke(Method.java:512)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.
java:847)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:605)
04-12 22:06:06.885: E/AndroidRuntime(7937):   at
dalvik.system.NativeStart.main(Native Method)
```

由此可见，只加载到第 8 张图片，程序就会报错：

```
java.lang.OutOfMemoryError: bitmap size exceeds VM budget
```

通过上面的例子可以看出：使用 Drawable 保存图片对象，会占用更小的内存空间。而使用 Biamtp 对象，则会占用很大的内存空间。

8.2 加载 APK 文件和 DEX 文件

在 Android 系统中，对编译出来的 DEX 字节码和 APK 文件的加载过程，也进行了尽可能的优化。具体来说有如下两点优化工作：

- 对于预置应用：Android 会在系统编译后，生成优化文件，以 ODEX 后缀结尾，这样在发布时除 APK 文件(不包含 DEX)外，还有一个相应的 ODEX 文件。
- 对于非预置应用：在运行前，Android 会优化 DEX 文件，在第一次启动应用时，执行文件的 DEX 被优化成 DEY 文件并放在/data/dalvik-cache 目录中。如果应用的 APK 文件不发生变化，DEX 文件不会被重新生成，加快了以后的启动速度。加载过程如图 8-4 所示。

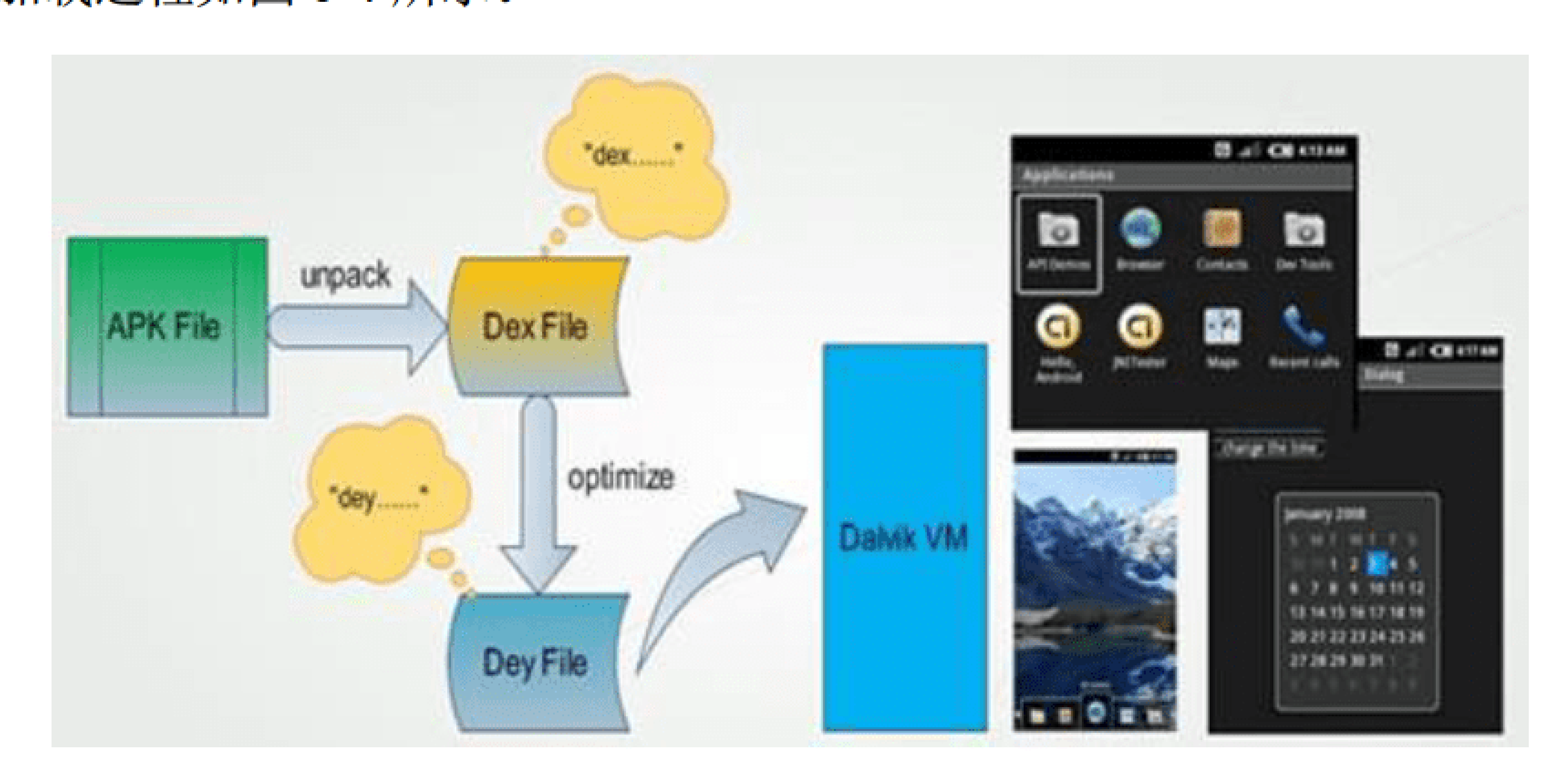


图 8-4 加载过程

DEX 文件由 header、string_ids、type_ids、proto_ids、field_ids、method_ids、class_defs、data 等几部分构成。图 8-5 显示了这几部分内容在 DEX 文件中的布局。

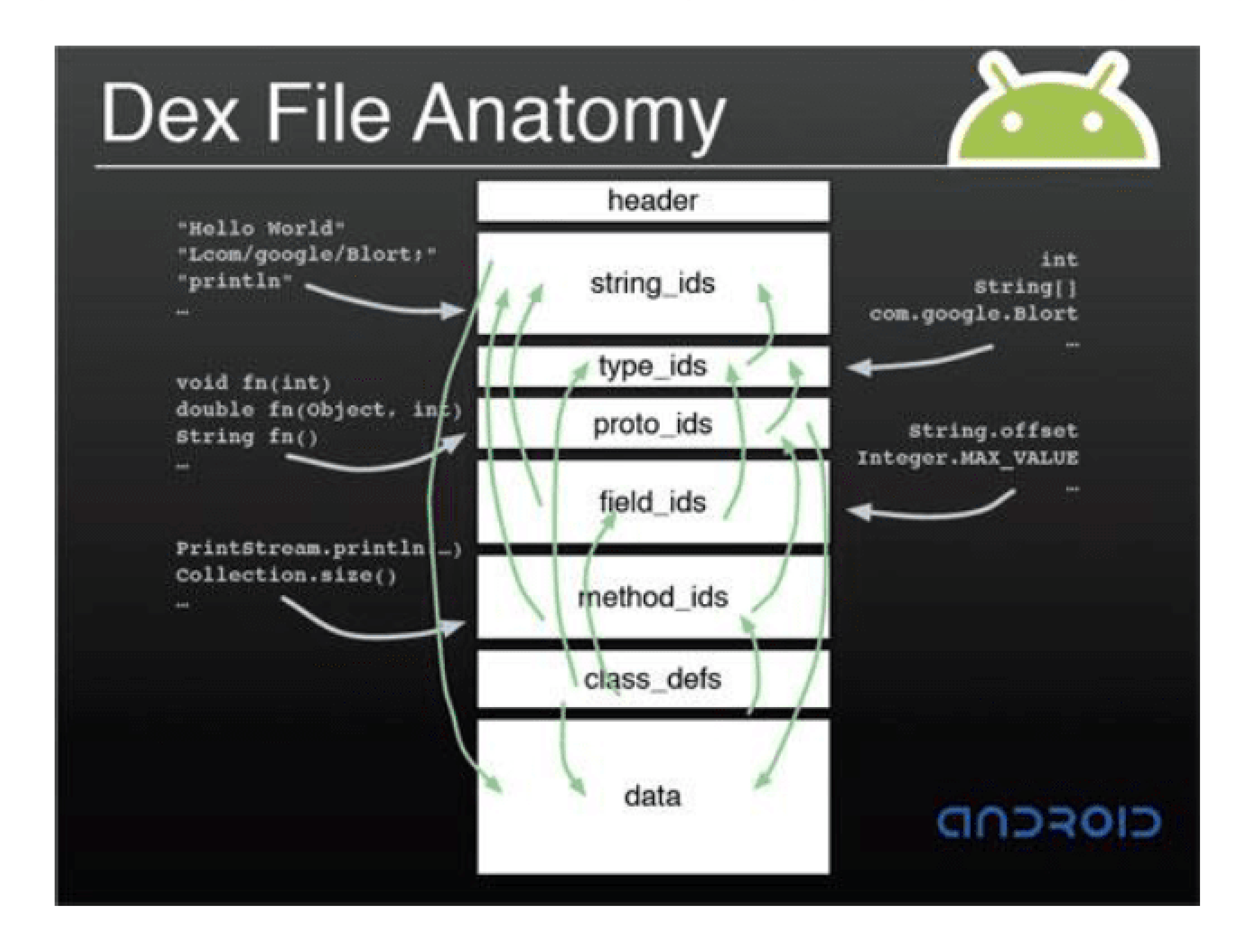


图 8-5 在 DEX 文件中的布局

在 Java 中，每一个类会被编译成相应的 CLASS 文件，一个应用会定义若干个类，这就导致同一个应用的多个 CLASS 文件中会存在冗余信息。而在 Android 中，“dx”工具会将同一个应用的所有 CLASS 文件内容整合到一个 DEX 文件中，这样就减小了整体的文件尺寸，I/O 操作也提高了类的查找速度。原来每个 CLASS 文件中的常量池，在 DEX 文件中由一个常量池来统一管理。“dx”工具整合 CLASS 文件的过程如图 8-6 所示。

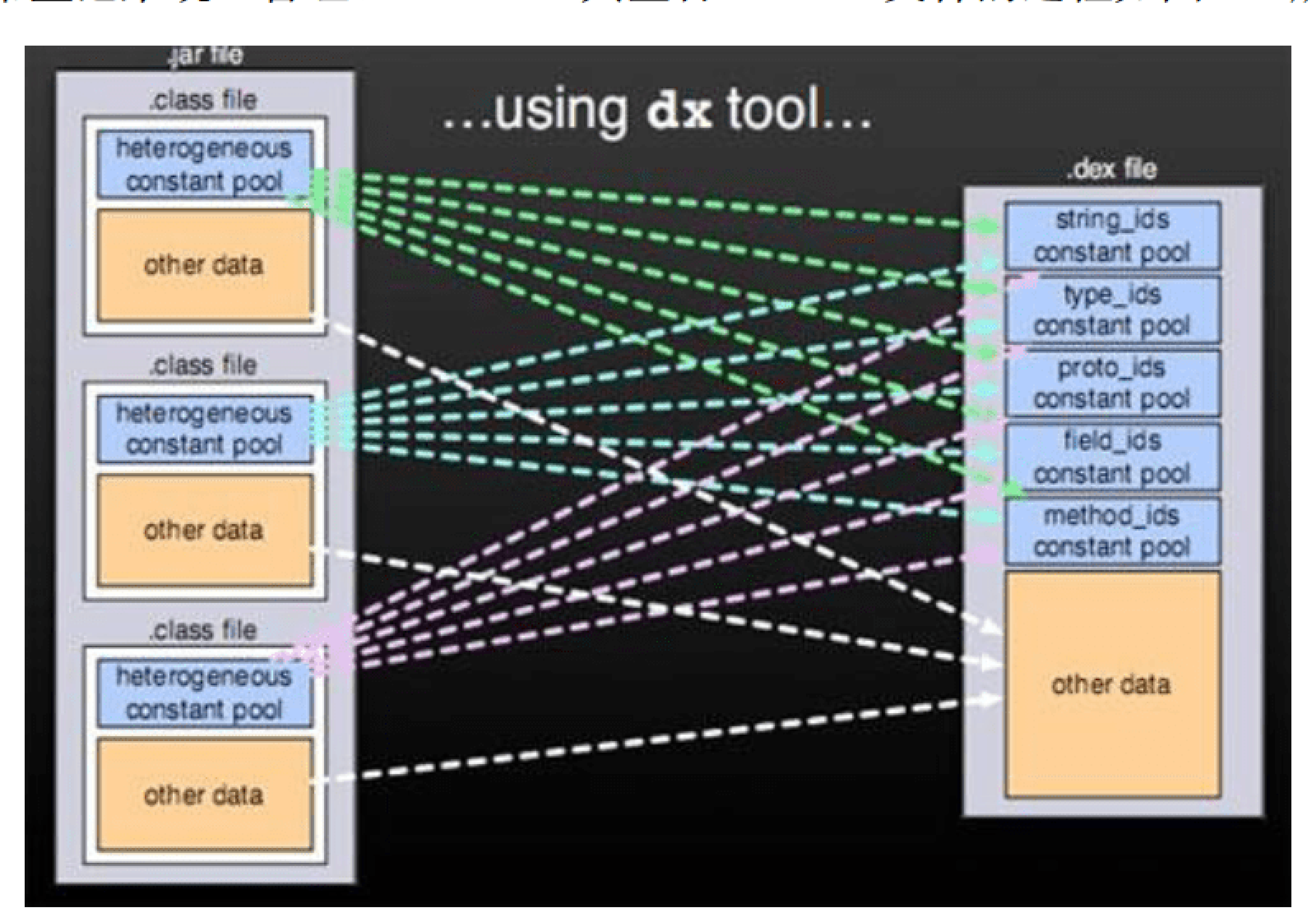


图 8-6　“dx”工具整合 CLASS 文件的过程

具体到 DEX 文件，经过“dx”工具优化后的内部逻辑如图 8-7 所示。

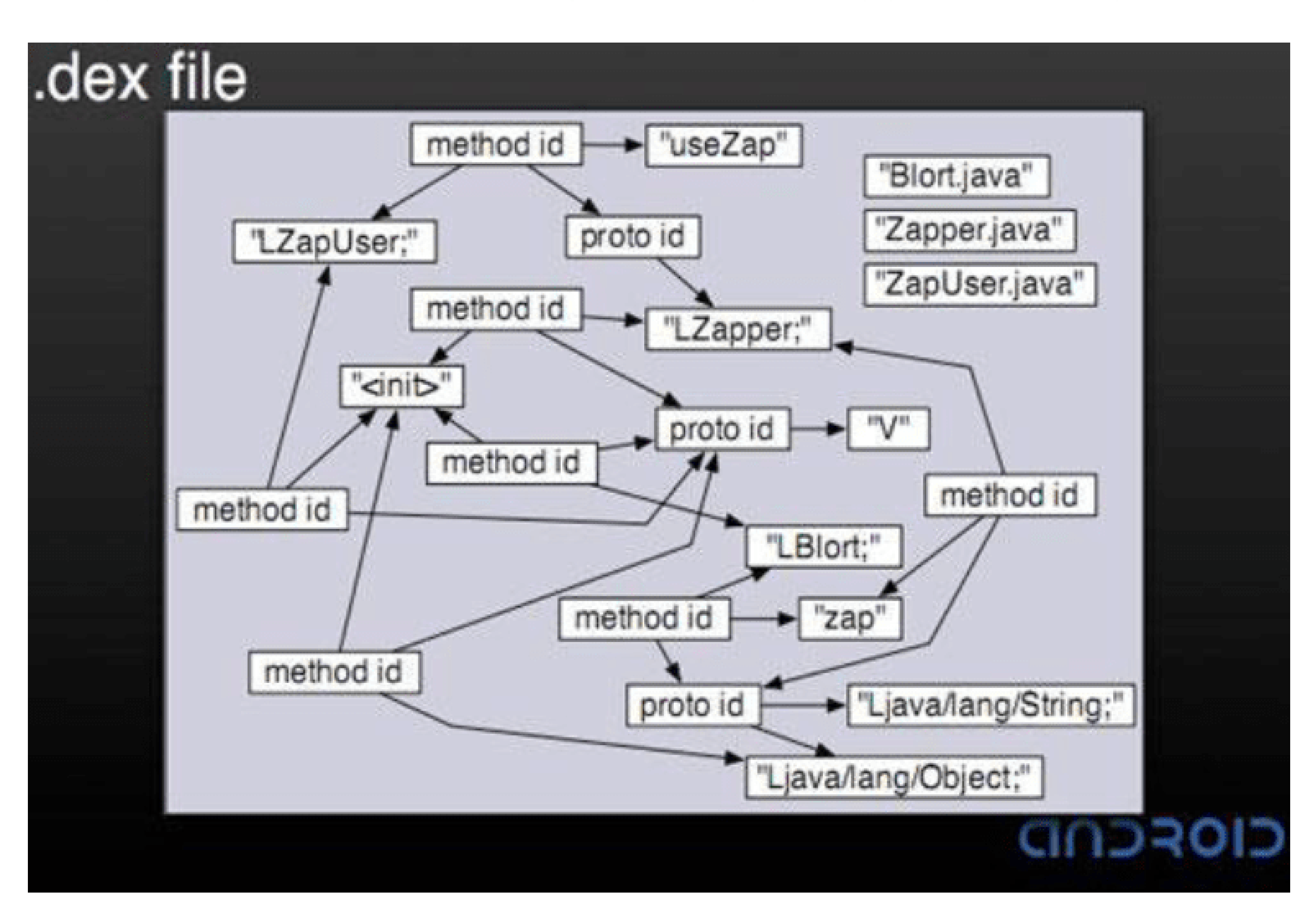


图 8-7　经过“dx”工具优化后的内部逻辑

8.2.1　APK 文件介绍

APK 是 Android Package 的缩写，即 Android 安装包。APK 是类似 Symbian Sis 或 Sisx

的文件格式。通过将 APK 文件直接传到 Android 模拟器或 Android 手机中执行即可安装。

(1) APK 文件的结构

APK 文件和 SIS 一样最终把 Android SDK 编译的工程打包成一个安装程序文件格式为 APK。APK 文件其实是 zip 格式，但后缀名被修改为 APK，通过 UnZip 解压后，可以看到 Dex 文件，Dex 是 Dalvik VM executes 的缩写，即 Android Dalvik 执行程序，并非 Java ME 的字节码而是 Dalvik 字节码。一个 APK 文件结构为：

- “res\”：存放资源文件的目录；
- AndroidManifest.xml：程序全局配置文件；
- classes.dex：Dalvik 字节码；
- resources.arsc：编译后的二进制资源文件。

经过总结后我们发现，Android 在运行一个程序时首先需要 UnZip，这样做对于程序的保密性和可靠性不是很高，通过 dexdump 命令可以反编译。在 Android 平台中，Dalvik VM 的执行文件被打包为 APK 格式，最终运行时加载器会解压，然后获取编译后的 androidmanifest.xml 文件中的 permission 分支相关的安全访问。但是这样仍然存在很多安全限制，如果将 APK 文件传到“/system/app”文件夹下，会发现执行是不受限制的。最终我们平时安装的文件可能不是这个文件夹，而在 Android ROOM 中系统的 APK 文件默认会放入这个文件夹，它们拥有着 ROOT 权限。

(2) 下载 APK 应用程序

我们可以从哪里取得好用的 Android APK 应用程序，并安装到 Android 手机上呢？对拥有 G1 实体手机的使用者而言，Android Market 就是最佳的地方，只要使用手机内应用程序列表的 Market 程序，就可以直接连接到 Android Market，而点选喜爱的应用程序后，就会直接下载并安装到 G1 手机上。不过对使用 Android 仿真器的使用者而言，就没有如此方便了，Android 仿真器并没有 Android Market 这个应用程序，只能使用内附的浏览器浏览 Android Market，为何说是浏览呢？因为 Android Market 不是采用通用网页浏览方式来下载文件，虽然可以使用常见的浏览器看到 Android Market 上的应用程序，但是没有办法下载到 Android 仿真器或一般的计算机上，原因是 Android Market 采用特有的网页 API，使用 native UI 的方式来访问，唯有通过内建在 G1 手机内的 Market 应用程序，才能下载 Android Market 网页中的应用程序，并自动安装到 G1 手机上。

所以 Android 仿真器的使用者，只好浏览该网页上的应用程序，然后通过搜索引擎去找找看有没有开发人员将应用程序放到 Android Market 之后，还另外将 APK 文件放置在一般网页上了。到此为止，使用 Android 仿真器的您，也不要这么灰心，因为有太多的人遇到同样的问题，就会生成很多 Android 应用程序网页，您可以浏览这些网页并把上面的 APK 文件下载到一般计算机上，再安装到 Android 仿真器上。

8.2.2 DEX 文件介绍和优化

DEX 即 Android Dalvik 执行程序，Google 在新发布的 Android 平台上使用了自己的 Dalvik 虚拟机来定义。这种虚拟机执行的并非 Java 字节码，而是另一种字节码：dex 格式的字节码。在编译 Java 代码之后，通过 Android 平台上的工具可以将 Java 字节码转换



成 Dex 字节码。这个 DalvikVM 针对手机程序程式的 CPU 进行了优化处理，可以同时执行许多 VM，而不会占用太多 Resource(资源)。

对于 Android DEX 文件进行优化，需要注意的一点是 DEX 文件的结构是紧凑的，但是我们还是要想方设法地提高程序的运行速度，我们就仍然需要对 DEX 文件进行进一步优化。

调整所有字段的字节序(LITTLE_ENDIAN)和对齐结构中的没一个域 验证 DEX 文件中的所有类 对一些特定的类进行优化，对方法里的操作码进行优化 。优化后的文件大小会有所增加，应该是原 Android DEX 文件的 1～4 倍。优化发生的时机有两个：对于预置应用，可以在系统编译后，生成优化文件，以 ODEX 结尾。

这样在发布时除 APK 文件(不包含 DEX)以外，还有一个相应的 Android DEX 文件；对于非预置应用，包含在 APK 文件里的 DEX 文件会在运行时被优化，优化后的文件将被保存在缓存中。每一个 Android 应用都运行在一个 Dalvik 虚拟机实例里，而每一个虚拟机实例都是一个独立的进程空间。虚拟机的线程机制，内存分配和管理，Mutex 等等都是依赖底层操作系统而实现的。

所有 Android 应用的线程都对应一个 Linux 线程，虚拟机因而可以更多的依赖操作系统的线程调度和管理机制。不同的应用在不同的进程空间里运行，加之对不同来源的应用都使用不同的 Linux 用户来运行，可以最大程度的保护应用的安全和独立运行。

Zygote 是一个虚拟机进程，同时也是一个虚拟机实例的孵化器，每当系统要求执行一个 Android 应用程序，Zygote 就会 FORK 出一个子进程来执行该应用程序。这样做的好处显而易见：Zygote 进程是在系统启动时产生的，它会完成虚拟机的初始化，库的加载，预置类库的加载和初始化等等操作，而在系统需要一个新的虚拟机实例时。

Zygote 通过复制自身，最快速的提供个系统。另外，对于一些只读的系统库，所有虚拟机实例都和 Zygote 共享一块内存区域，大大节省了内存开销。Android 应用开发和 Dalvik 虚拟机 Android 应用所使用的编程语言是 Java 语言，和 Java SE 一样，编译时使用 Sun JDK 将 Java 源程序编程成标准的 Java 字节码文件(.class 文件)。

而后通过工具软件 DX 把所有的字节码文件转成 Android DEX 文件(classes.dex)。最后使用 Android 打包工具(aapt)将 DEX 文件，资源文件以及 AndroidManifest.xml 文件(二进制格式)组合成一个应用程序包(APK)。应用程序包可以被发布到手机上运行。

8.2.3　Android 类动态加载技术实现加密优化

在加载 APK 文件和 DEX 文件时，使用了类动态加载技术。在 Android 应用开发过程中，通常常规的开发方式和代码架构就能满足我们的普通需求。但是有些特殊问题，常常引发我们进一步的沉思。考虑下面的问题：

- 如何开发一个可以自定义控件的 Android 应用？就像 Eclipse 一样，可以动态加载插件？
- 如何让 Android 应用执行服务器上的不可预知的代码？
- 如何对 Android 应用加密，而只在执行时自解密，从而防止被破解？

熟悉 Java 技术的读者会想到，我们需要使用类加载器灵活的加载执行的类。这在 Java

中已经算是一项比较成熟的技术了，但是在 Android 应用中，我们都还比较陌生。

(1) 类加载机制

Dalvik 虚拟机如同其他 Java 虚拟机一样，在运行程序时首先需要将对应的类加载到内存中。而在 Java 标准的虚拟机中，类加载可以从 class 文件中读取，也可以是其他形式的二进制流。因此，我们常常利用这一点，在程序运行时手动加载 Class，从而达到代码动态加载执行的目的。

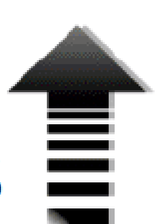
但是 Dalvik 虚拟机毕竟不算是标准的 Java 虚拟机，因此在类加载机制方面它们有相同的地方，也有不同之处。我们必须区别对待。例如，在使用标准 Java 虚拟机时，经常自定义继承自 ClassLoader 的类加载器。然后通过 defineClass()方法来从一个二进制流中加载 Class，但是这在 Android 里是行不通的，这一点可以从 Android 源码知道。Android 中 ClassLoader 的 defineClass()方法，具体是调用 VMClassLoader 的 defineClass()本地静态方法。而这个本地方法除了抛出一个“UnsupportedOperationException”异常之外，什么都没做，甚至连返回值都为空。下面是演示代码：

```
static void Dalvik java lang VMClassLoader defineClass(const u4*
args,JValue* pResult)
{
    Object* loader = (Object*) args[0];
    StringObject* nameObj = (StringObject*) args[1];
    const u1* data = (const u1*) args[2];
    int offset = args[3];
    int len = args[4];
    Object* pd = (Object*) args[5];
    char* name = NULL;
    name = dvmCreateCstrFromString(nameObj);
    LOGE("ERROR: defineClass(%p, %s, %p, %d, %d, %p)\n",
        loader, name, data, offset, len, pd);
    dvmThrowException("Ljava/lang/UnsupportedOperationException;",
        "can't load this type of class file");
    free(name);
    RETURN VOID();
}
```

(2) Dalvik 虚拟机类的加载机制

那如果在 Dalvik 虚拟机里，ClassLoader 不好使，我们该如何实现动态加载类呢？Android 为我们从 ClassLoader 派生出了两个类：DexClassLoader 和 PathClassLoader。其中需要特别说明的是，PathClassLoader 中如下被注释掉的代码：

```
/* --this doesn't work in current version of Dalvik--
    if (data != null) {
        System.out.println("--- Found class " + name
            + " in zip[" + i + "] '" + mZips[i].getName() + "'");
        int dotIndex = name.lastIndexOf('.');
        if (dotIndex != -1) {
            String packageName = name.substring(0, dotIndex);
            synchronized (this) {
               Package packageObj = getPackage(packageName);
```



```
                if (packageObj == null) {
                    definePackage(packageName, null, null,
                            null, null, null, null, null);
                }
            }
        }
        return defineClass(name, data, 0, data.length);
    }
*/
```

这可以从另一方面证明了 defineClass()函数在 Dalvik 虚拟机中被“阉割”了。而在这两个继承自 ClassLoader 的类加载器，本质上是重载了 ClassLoader 的 findClass 方法。在执行 loadClass 时可以参照 ClassLoader 的部分源码：

```
protected Class<?> loadClass(String className, boolean resolve)
throws ClassNotFoundException {
Class<?> clazz = findLoadedClass(className);
    if (clazz == null) {
        try {
            clazz = parent.loadClass(className, false);
        } catch (ClassNotFoundException e) {
            // Don't want to see this.
        }
        if (clazz == null) {
            clazz = findClass(className);
        }
    }
    return clazz;
}
```

由此可见，DexClassLoader 和 PathClassLoader 都属于符合双亲委派模型的类加载器(因为它们没有重载 loadClass 方法)。也就是说，它们在加载一个类之前，会检查自己以及自己以上的类加载器是否已经加载了这个类。如果已经加载过了，则会直接将之返回，而不会重复加载。

DexClassLoader 和 PathClassLoader 其实都是通过类 DexFile 实现类加载功能的。这里需要顺便提一下的是，Dalvik 虚拟机识别的是 dex 文件，而不是 class 文件。因此，我们供类加载的文件也只能是 dex 文件，或者包含有 dex 文件的.apk 或.jar 文件。

PathClassLoader 是通过构造函数 new DexFile(path)来生成生 DexFile 对象的；而 DexClassLoader 则是通过其静态方法 loadDex(path, outpath, 0)得到 DexFile 对象的。这两者的区别在于 DexClassLoader 需要提供一个可写的 outpath 路径，用来释放.apk 包或者.jar 包中的 dex 文件。也就是说，PathClassLoader 不能主动从 zip 包中释放出 dex，因此只支持直接操作 dex 格式文件，或者已经安装的 apk(因为已经安装的 apk 在 cache 中存在缓存的 dex 文件)。而 DexClassLoader 可以支持.apk、.jar 和.dex 文件，并且会在指定的 outpath 路径释放出 dex 文件。

当 PathClassLoader 在加载类时，调用的是 DexFile 的 loadClassBinaryName，而 DexClassLoader 调用的是 loadClass。所以在使用 PathClassLoader 时，类的全名需要用“/”替换“.”。

(3) 具体操作

在具体操作时，可能需要使用到的工具有：javac、dx、eclipse 等。其中在使用 dx 工具时，最好指明：--no-strict，因为 class 文件的路径可能不匹配。当加载好类后，通常可以通过 Java 反射机制来使用这个类。但是这样做的效率相对不高，而且老用反射代码也比较复杂凌乱。更好的做法是定义一个 interface，并将这个 interface 写进容器端。待加载的类，继承自这个 interface，并且有一个参数为空的构造函数，以使我们能够通过 Class 的 newInstance 方法产生对象。然后将对象强制转换为 interface 对象，于是就可以直接调用成员方法了。

(4) 代码加密

在加密代码时，最初设想将 dex 文件加密，然后通过 JNI 将解密代码写在 Native 层。解密之后直接传入二进制流，再通过 defineClass 将类加载到内存中。但是由于不能直接使用 defineClass，而必须传文件路径给 Dalvik 虚拟机内核，因此解密后的文件需要写到磁盘上，增加了被破解的风险。

Dalvik 虚拟机内核仅支持从 dex 文件加载类的方式是不灵活的，由于没有非常深入的研究内核，我不能确定是 Dalvik 虚拟机本身不支持还是 Android 在移植时将其阉割了。不过坚信的是，Dalvik 或 Android 开源项目都正在向能够支持 raw 数据定义类的方向努力。

在 RawDexFile 出来之前，我们只能使用这种存在一定风险的加密方式。我们需要注意释放的 dex 文件路径及权限管理。另外在类加载完毕之后，除非出于其他目的，否则应该马上删除临时的解密文件。

8.3 SD 卡优化

SD 卡作为手机的扩展存储设备，在手机中充当硬盘角色，可以让我们手机存放更多的数据以及多媒体等大体积文件。因此查看 SD 卡的内存就跟我们查看硬盘的剩余空间一样，是我们经常操作的一件事，那么在 Android 开发中，我们如何能获取 SD 卡的内存容量呢？

(1) 首先，要获取 SD 卡上面的信息，必须先对 SD 卡有访问的权限，因此第一件事就是需要添加访问扩展设备的权限。

```
<uses-permission
android:name="android.permission.WRITE EXTERNAL STORAGE">
</uses-permission>
```

(2) 然后需要判断手机上面 SD 卡是否插好，如果有 SD 卡的情况下才可以访问得到并获取到它的相关信息，当然以下这个语句需要用 if 做判断。

```
Environment.getExternalStorageState().equals(Environment.MEDIA_MOUNTED)
```

取得 SD 卡文件的路径：

```
    File path = Environment.getExternalStorageDirectory();
    StatFs statfs = new StatFs(path.getPath());
```

获取 block 的 SIZE：

```
    long blocSize = statfs.getBlockSize();
```

获取 BLOCK 数量：

```
long totalBlocks = statfs.getBlockCount();
```

获取空闲的 Block 的数量：

```
long availaBlock = statfs.getAvailableBlocks();
```

计算总空间大小和空闲的空间大小

```
    /**
    * 取得空闲 sd 卡空间大小
    * @return
    */
    public long getAvailaleSize(){
    File path = Environment.getExternalStorageDirectory(); //取得 sdcard 文
件路径
    StatFs stat = new StatFs(path.getPath());
    /*获取 block 的 SIZE*/
    long blockSize = stat.getBlockSize();
    /*空闲的 Block 的数量*/
    long availableBlocks = stat.getAvailableBlocks();
    /* 返回 bit 大小值*/
    return availableBlocks * blockSize/1024/1024;
    //(availableBlocks * blockSize)/1024 KIB 单位
    //(availableBlocks * blockSize)/1024 /1024 MIB 单位
    }
    /**
    * SD 卡大小
    * @return
    */
    public long getAllSize(){
    File path = Environment.getExternalStorageDirectory();
    StatFs stat = new StatFs(path.getPath());
    /*获取 block 的 SIZE*/
    long blockSize = stat.getBlockSize();
    /*块数量*/
    long availableBlocks = stat.getBlockCount();
    /* 返回 bit 大小值*/
    return availableBlocks * blockSize/1024/1024;
    }
```

(1) 加载优化

Android 的图片浏览器等多媒体应用可以加载整个 SD 卡内的所有图像，在加载前会把数据做成数据库，不用每次扫描，这大大加快了启动速度。事实上扫描操作是通过 MediaScanner 来实现的，目前支持的文件类型在 MediaFile.java 中定义。主要包括音频、MIDI、视频、图片、播放列表等。MediaScannerService 服务的启动仅在收到如下权限后才会启动。

- android.intent.action.BOOT_COMPLETED
- android.intent.action.MEDIA_MOUNTED
- android.intent.action.MEDIA_SCANNER_SCAN_FILE

当然，在 SD 卡容量较大且文件较多时，MediaScannerService 服务将会运行一段不短的时间，这对电池的持续能力会造成一定的影响，尤其是在电池技术始终不能有显著突破的前提下。

(2) 分区优化

在将 SD 卡分区时，通常把第一分区的簇的大小设置为 16k 或者更大之后，都会得到更高的 PC 测试得分，而且手机也会增加流畅度。这是因为 fat/fat32/vfat 系统采用扇区+簇的方式来存储文件，一个扇区一般是 512 字节，一个簇就是一组扇区的集合。在默认状态下，4G 以上的 fat32 分区应该是每簇 16 个扇区，也就是 16×512=8K 字节，这个字节对于 Android 的使用来说偏小了，当然也会提高些空间利用率。通过调大簇的大小，例如调到 64 个扇区(32K)，可以提高大文件的存取效率。假设一个文件的大小是 1024K，如果是 8K 的簇则最坏情况需要 1024/8=128 次 IO。如果是 32K 的簇，则最坏情况只需要 1024/32=32 次 IO，当然实际的 IO 次数可能比这些都少，因为操作系统有自己的优化方法，会尽量多读一些进来，最坏情况指的是 1024K 的数据真的被分别存在 128 个互不相邻的簇上，这样就是真的 128 次 IO 了。因此更大的簇对于大文件是有非常好的优化效果的，现在我们日常用的文件其实大部分都大于 1024K 了，比如一个 MP3 至少也要 3M 才算可听，而导航数据就更大了。因此尽量使用更大的簇是很有必要的。

所以在优化分区时，建议在格式化 SD 卡第一分区(fat32)的时候设置簇大小为 32K，其实最高可以到 64K，但是 64K 是 fat32 设计的极限，从软件角度来说在极限状态运行是不可靠的。因此使用较低一档的大小，格式化之后把数据复制回去。

为什么提高了 fat32 分区的效率就会提高手机的整体效率呢？这是因为这两个分区是在一个硬件上，如果 fat32 占用的 IO 负载大，则 Ext 分区分到的 IO 带宽自然就小了，而 Android 手机在第一次运行的时候其实是非常频繁地访问 fat32 分区的，因为“Media Scaner”在做数据搜集扫描工作时，为 Android 特有的手机全局搜索准备数据。因此，对 fat32 的优化可以提高整个手机的运行效率。当然我们可以等 Media Scaner 扫描完后再用手机，这样会很流畅，那时候手机流畅度就跟 fat32 是否优化无关了。

8.4 Android 的虚拟机优化

在虚拟机和原生库层面，Android 同样进行了很多的优化。在本节的内容中，将详细讲解 Android 虚拟机优化的基本知识。

8.4.1 Android 虚拟机概述

虚拟机中指令的解释时间主要分为 3 个方面，分别是分发指令、访问运算数、执行运算。其中“分发指令”这个环节对性能的影响最大，为了加快运行速度，必须提高分发指

令的速度。

与传统的 Java 虚拟机基于栈不同，Dalvik 是基于寄存器的。基于寄存器的虚拟机实现，虽然在硬件通用性上稍逊一筹，但是数据处理速度却有明显的改善，可以更为有效地减小冗余指令的分发和减小内存的读写访问。

Dalvik 虚拟机针对移动终端所做的优化，使得其不需要很快的 CPU 速度和大量的内存空间。根据 Google 的测算，Android 的早期版本只需要 64MB 的 RAM 即可使系统正常运转，其中 24MB 被用于底层系统的初始化和启动，另外 20MB 被用于高层启动、高层服务。随着 Android 版本的不断升级和应用功能的扩展，Android 对内存的消耗也在逐渐增加。

另外需要注意的是，Dalvik 并不是按照 Java 虚拟机的规范来实现的，两者并不兼容。Java 虚拟机运行的是 Java 字节码，而 Dalvik 虚拟机运行的则是其专有的 DEX(Dalvik Executable)字节码。

在 Java SE 程序中，Java 类会被编译成一个或者多个字节码文件(.class)，然后打包成 JAR 文件。在执行期间，Java 虚拟机会从 JAR 文件抽取相应的 CLASS 文件并从中读取指令和数据。而 Android 虽然也是基于 Java 语言进行编程的，但是在编译成 CLASS 文件后，Android 会通过“dx”工具将应用所有的 CLASS 文件转换一个 DEX 文件，接着将 DEX 和应用的其他如资源文件等一起打包构成 APK 文件，而后 Dalvik 虚拟机会从其中读取指令和数据。图 8-8 显示了 Android 的编译过程。

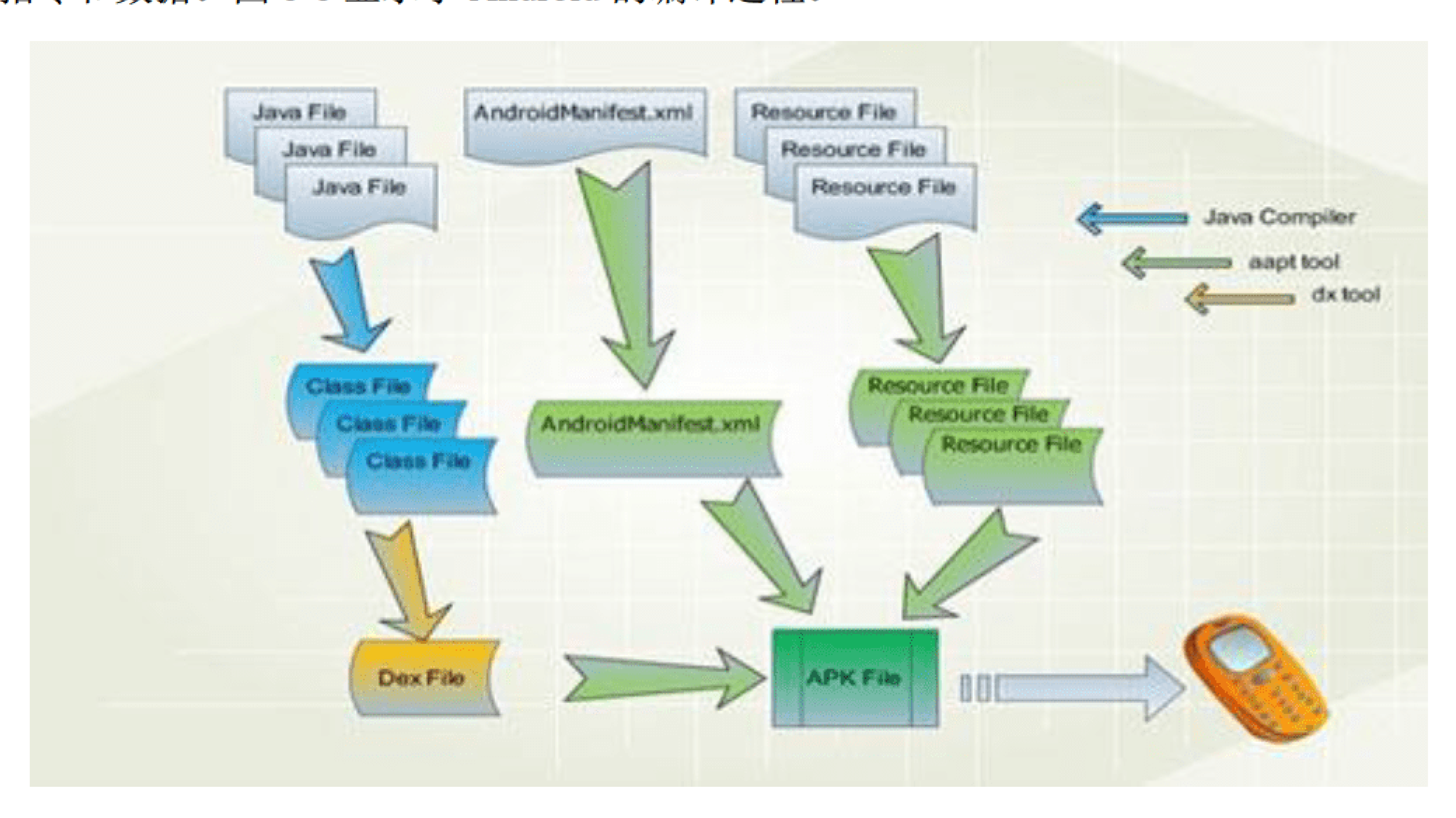


图 8-8　Android 的编译过程

Dalvik 虚拟机的主要特征如下。

- 专有的 DEX 字节码。
- 支持新的操作码。
- 文件结构非常简洁。
- 使用等长的指令。
- 借以提升解析速度。

- 尽量扩大只读结构的大小。
- 借以提高跨进程的数据共享比例。

8.4.2 平台优化——ARM 的流水线技术

Android 虚拟机充分挖掘了 CPU 的性能，针对 armv5te 进行了优化，充分利用 armv5te 的执行流水线来提高执行的效率。在 Android 刚诞生的时候，虽然支持 ARM CPU，其实实际上只支持 armv5te 的指令集，因为 Android 系统专门为 armv5te 进行了优化，充分利用 armv5te 的执行流水线来提高执行的效率，这也是在 500M 的三星 2440 运行效果不是很好，而在 200M 的 OMAP CPU 上运行比较流畅的原因了。所以在最新的代码中有专门针对 x86 和 armv4 的优化部分。

1. 什么是流水线技术

流水线技术通过多个功能部件并行工作来缩短程序执行时间，提高了处理器核的效率和吞吐率，从而成为微处理器设计中最为重要的技术之一。

ARM 7 的三级流水线在执行单元完成了大量的工作，包括与操作数相关的寄存器和存储器读写操作、ALU 操作以及相关器件之间的数据传输。执行单元的工作往往占用多个时钟周期，从而成为系统性能的瓶颈。ARM9 采用了更为高效的五级流水线设计，增加了两个功能部件分别访问存储器并写回结果，且将读寄存器的操作转移到译码部件上，使流水线各部件在功能上更平衡；同时其哈佛架构避免了数据访问和取指的总线冲突。

然而不论是三级流水线还是五级流水线，当出现多周期指令、跳转分支指令和中断发生的时候，流水线都会发生阻塞，而且相邻指令之间也可能因为寄存器冲突导致流水线阻塞，降低流水线的效率。本节在对流水线原理及运行情况详细分析的基础上，研究通过调整通过多个功能部件并行工作来缩短程序执行时间，提高处理器核的效率和吞吐率。

ARM 7 处理器核使用了典型三级流水线的冯·诺伊曼结构，ARM 9 系列则采用了基于五级流水线的哈佛结构。通过增加流水线级数简化了流水线各级的逻辑，进一步提高了处理器的性能。

2. ARM 7 流水线技术

ARM 7 系列处理器中每条指令分取指、译码、执行三个阶段，分别在不同的功能部件上依次独立完成。取指部件完成从存储器装载一条指令，通过译码部件产生下一周期数据路径需要的控制信号，完成寄存器的解码，再送到执行单元完成寄存器的读取、ALU 运算及运算结果的写回，需要访问存储器的指令完成存储器的访问。流水线上虽然一条指令仍需 3 个时钟周期来完成，但通过多个部件并行，使得处理器的吞吐率约为每个周期一条指令，提高了流式指令的处理速度，从而可达到 0.9MIPS/MHz 的指令执行速度。

在三级流水线下，通过 R15 访问 PC(程序计数器)时会出现取指位置和执行位置不同的现象。这须结合流水线的执行情况考虑，取指部件根据 PC 取指，取指完成后 PC+4 送到 PC，并把取到的指令传递给译码部件，然后取指部件根据新的 PC 取指。因为每条指令 4 字节，故 PC 值等于当前程序执行位置+8。

3. ARM 9 流水线技术

ARM 9 系列处理器的流水线分为取指、译码、执行、访存、回写。取指部件完成从指令存储器取指；译码部件读取寄存器操作数，与三级流水线中不占有数据路径区别很大；执行部件产生 ALU 运算结果或产生存储器地址(对于存储器访问指令来讲)；访存部件访问数据存储器；回写部件完成执行结果写回寄存器。把三级流水线中的执行单元进一步细化，减少了在每个时钟周期内必须完成的工作量，进而允许使用较高的时钟频率，且具有分开的指令和数据存储器，减少了冲突的发生，每条指令的平均周期数明显减少。

4. 三级流水线运行情况分析

三级流水线在处理简单的寄存器操作指令时，吞吐率为平均每个时钟周期一条指令。但是在存在存储器访问指令、跳转指令的情况下会出现流水线阻断情况，导致流水线的性能下降。图 8-9 给出了流水线的最佳运行情况，图中的 MOV、ADD、SUB 指令为单周期指令。从 T1 开始，用 3 个时钟周期执行了 3 条指令，指令平均周期数(CPI)等于 1 个时钟周期。



图 8-9　ARM 7 单周期指令的最佳流水线

流水线中阻断现象也十分普遍，下面就各种阻断情况下的流水线性能进行详细分析。

(1) 带有存储器访问指令的流水线

对存储器的访问指令 LDR 就是非单周期指令，如图 8-10 所示。这类指令在执行阶段，首先要进行存储器的地址计算，占用控制信号线，而译码的过程同样需要占用控制信号线，所以下一条指令(第一个 SUB)的译码被阻断，并且由于 LDR 访问存储器和回写寄存器的过程中需要继续占用执行单元，所以下一条(第一个 SUB)的执行也被阻断。由于采用冯・诺伊曼体系结构，不能够同时访问数据存储器和指令存储器，当 LDR 处于访存周期的过程中时，MOV 指令的取指被阻断。因此处理器用 8 个时钟周期执行了 6 条指令，指令平均周期数(CPI)=1.3 个时钟周期。

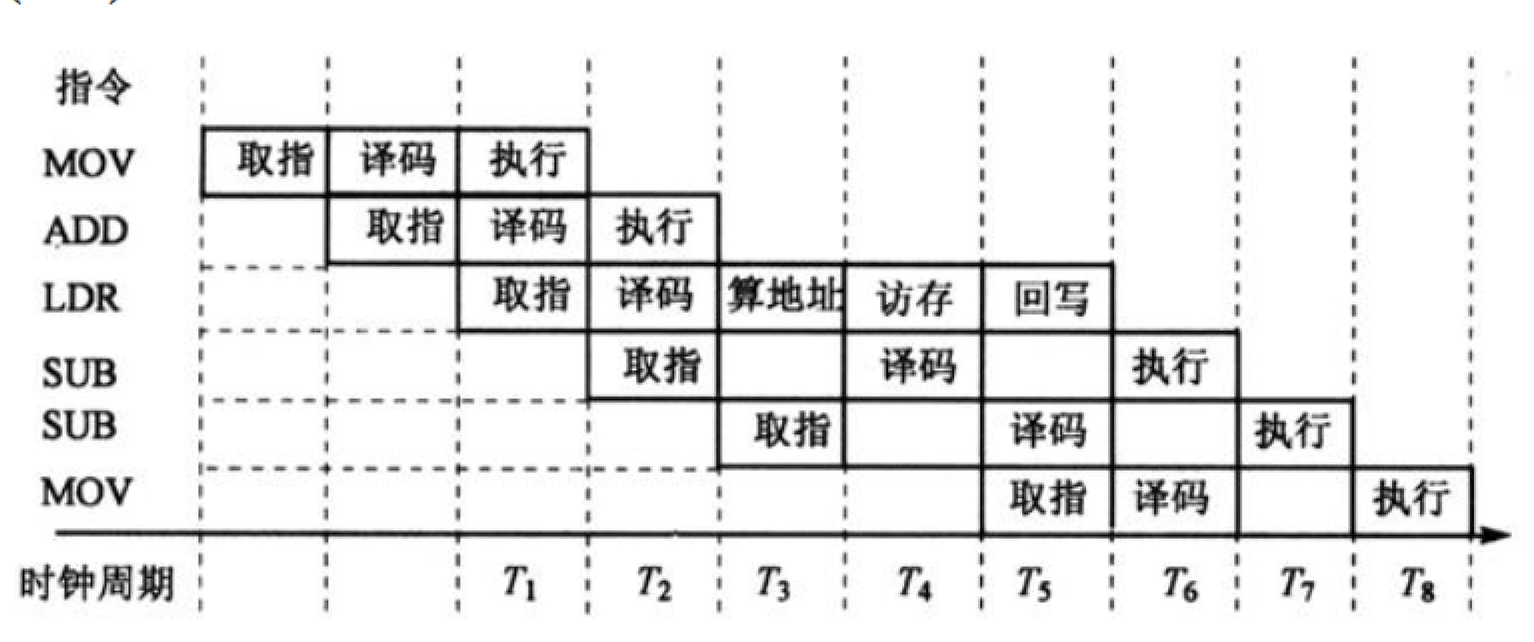


图 8-10　带有存储器访问指令的流水线

(2) 带有分支指令的流水线

当指令序列中含有具有分支功能的指令(如 BL 等)时，流水线也会被阻断，如图 8-11 所示。分支指令在执行时，其后第 1 条指令被译码，其后第 2 条指令进行取指，但是这两步操作的指令并不被执行。因为分支指令执行完毕后，程序应该转到跳转的目标地址处执行，因此在流水线上需要丢弃这两条指令，同时程序计数器就会转移到新的位置接着进行取指、译码和执行。此外还有一些特殊的转移指令需要在跳转完成的同时进行写链接寄存器、程序计数寄存器，如 BL 执行过程中包括两个附加操作——写链接寄存器和调整程序指针。这两个操作仍然占用执行单元，这时处于译码和取指的流水线被阻断了。

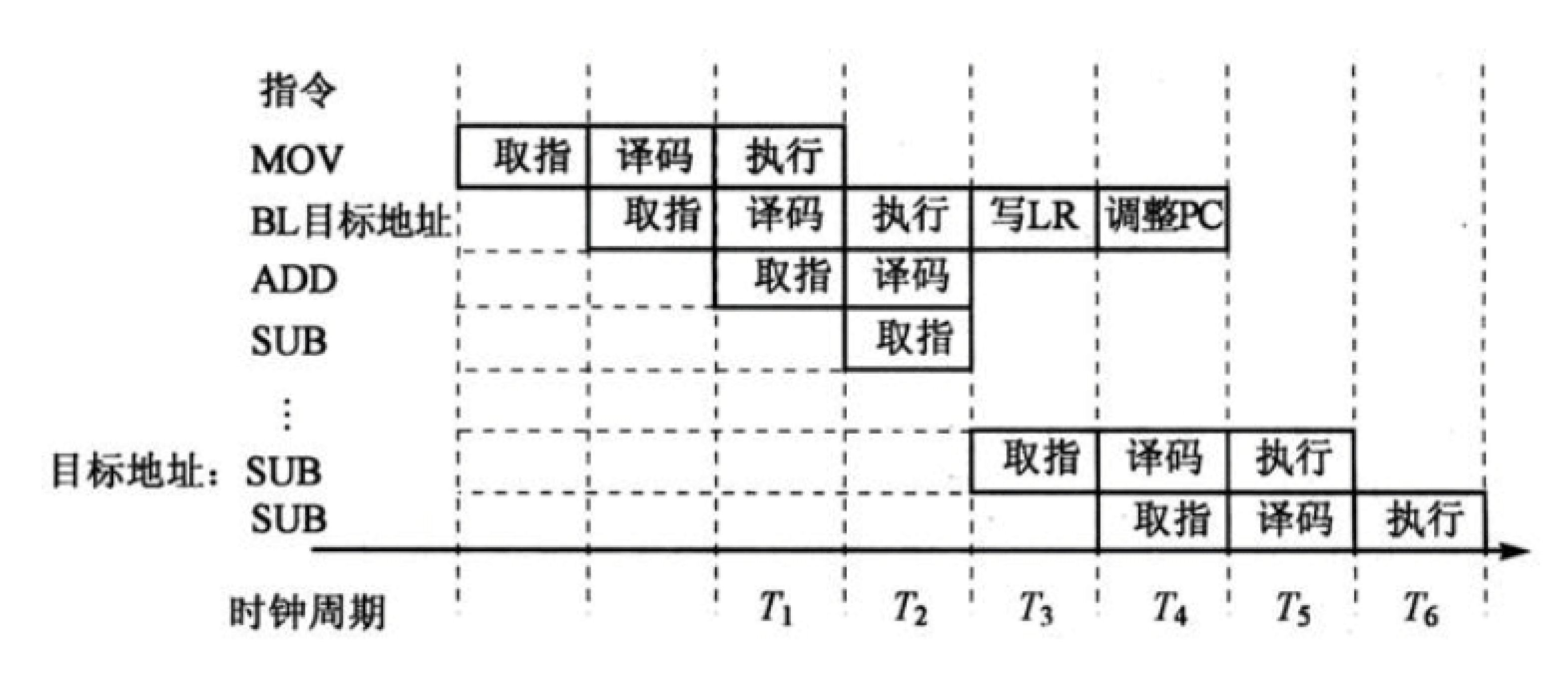


图 8-11　带有分支指令的流水线

5. 五级流水线技术

五级流水线只存在一种互锁，即寄存器冲突。

(1) 五级流水线互锁分析

读寄存器是在译码阶段，写寄存器是在回写阶段。如果当前指令(A)的目的操作数寄存器和下一条指令(B)的源操作数寄存器一致，B 指令就需要等 A 回写之后才能译码。这就是五级流水线中的寄存器冲突。如图 8-12 所示，LDR 指令写 R9 是在回写阶段，而 MOV 中需要用到的 R9 正是 LDR 在回写阶段将会重新写入的寄存器值，MOV 译码需要等待，直到 LDR 指令的寄存器回写操作完成。在当前处理器设计中，可以通过寄存器旁路技术对流水线进行优化，解决流水线的寄存器冲突问题。

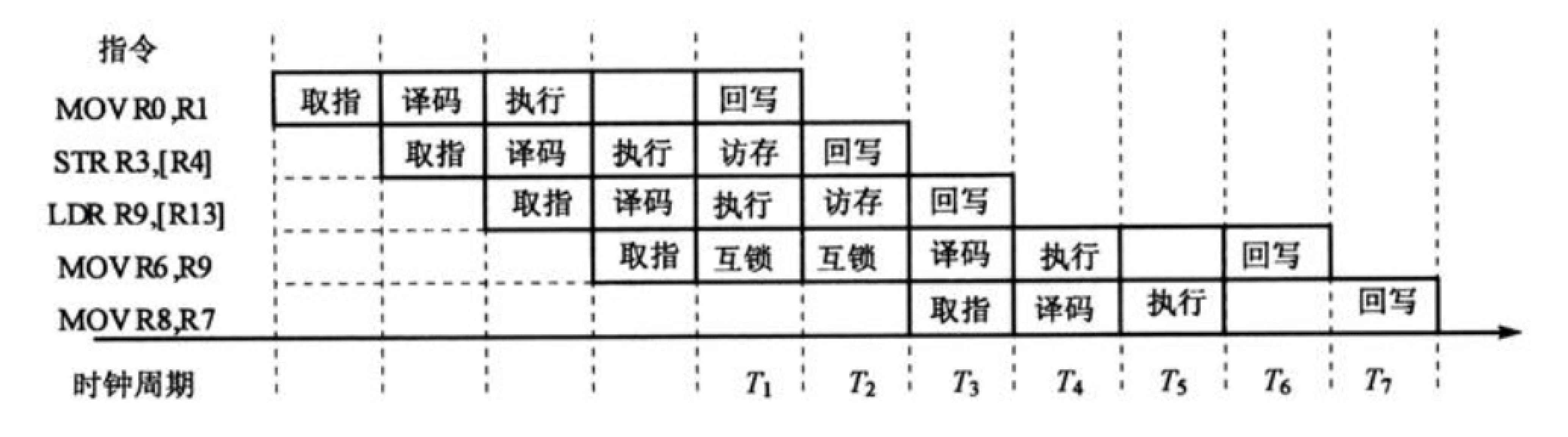


图 8-12　ARM 9 的五级流水线互锁

虽然流水线互锁会增加代码执行时间，但是为初期的设计者提供了巨大的方便，可以不必考虑使用的寄存器会不会造成冲突；而且编译器以及汇编程序员可以通过重新设计代码的顺序或者其他方法来减少互锁的数量。另外分支指令和中断的发生仍然会阻断五级流水线。

(2) 五级流水线优化

采用重新设计代码顺序在很多情况下可以很好地减少流水线的阻塞，使流水线的运行流畅。下面详细分析代码优化对流水线的优化和效率的提高。

假设要实现把内存地址 0x1000 和 Ox2000 处的数据分别拷贝到 0x8000 和 0x9000 处。

Ox1000 处的内容为：1，2，3，4，5，6，7，8，9，10

Ox2000 处的内容为：H，e，l，l，o，W，o，r，l，d

实现第一个拷贝过程的程序代码及指令的执行时空图如图 8-13 所示。

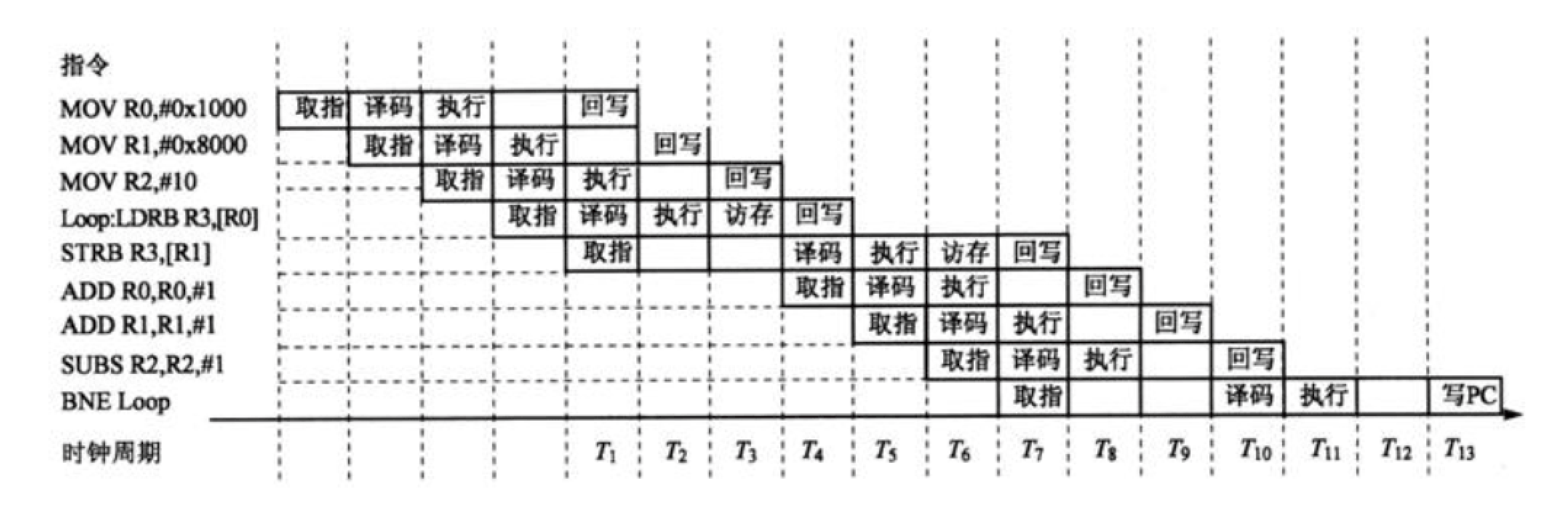


图 8-13 没有经过优化的流水线

全部拷贝过程由两个结构相同的循环各自独立完成，分别实现两块数据的拷贝，并且两个拷贝过程极为类似，分析其中一个即可。

在图 8-13 中，T1～T3 是 3 个单独的时钟周期；T4～T11 是一个循环，在时空图中描述了第一次循环的执行情况。在 T12 的时候写 LR 的同时，开始对循环的第一条语句进行取指，所以总的流水线周期数为 3+10×10+2×9=121。整个拷贝过程需要 121×2+2=244 个时钟周期完成。

考虑到通过减少流水线的冲突可以提高流水线的执行效率，而流水线的冲突主要来自寄存器冲突和分支指令，因此对代码作如下两方面调整：

- 将两个循环合并成一个循环能够充分减少循环跳转的次数，减少跳转带来的流水线停滞；
- 调整代码的顺序，将带有与邻近指令不相关的寄存器插到带有相关寄存器的指令之间，能够充分地避免寄存器冲突导致的流水线阻塞。

对代码调整和流水线的时空图如图 8-14 所示。

由此可见，在调整之后，T1～T5 是 5 个单独的时钟周期，T6～T13 是一个循环，同样在 T14 的时候 BNE 指令在写 LR 的同时，循环的第一条指令开始取指，所以总的 指令周期数为 5+10×10+2×9+2=125。

通过两段代码的比较可看出：调整之前整个拷贝过程总共使用了 244 个时钟周期，调整了循环内指令的顺序后，总共使用了 125 个时钟周期就完成了同样的工作，时钟周期减少了 119 个，缩短了 119/244=48.8%，效率提升十分明显。

代码优化前后执行周期数对比的情况如表 8-1 所示。

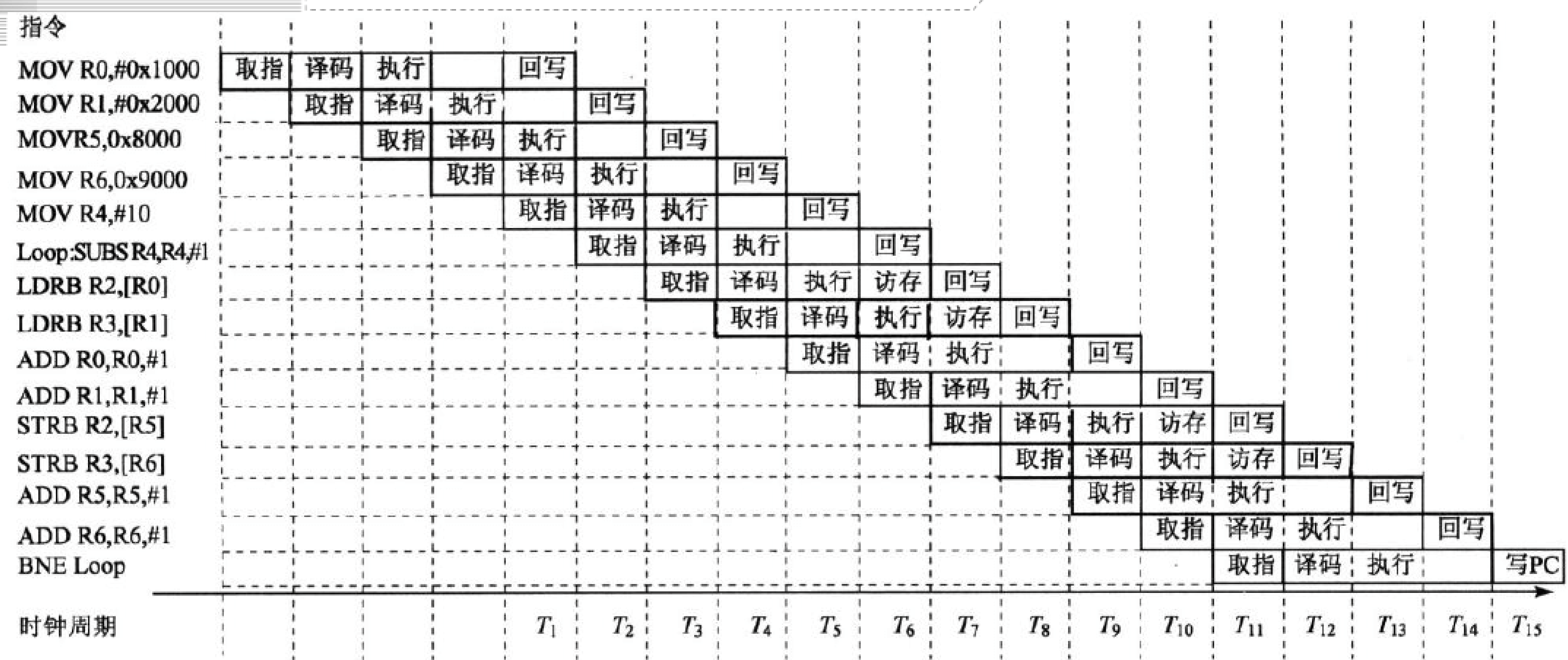


图 8-14　优化后的流水线

表 8-1　代码优化前后执行周期数对比

	优化前周期数	优化后周期数	提高比例/(%)
顺序语句	6	5	16.7
循环 1	118	60	49.2
循环 2	120	60	50
总周期数	244	125	48.8

所以流水线的优化问题主要应该从如下两个方面考虑。

- 通过合并循环等方式减少分支指令的个数，从而减少流水线的浪费；
- 通过交换指令的顺序，避免寄存器冲突造成的流水线停滞。

8.4.3　Android 对 C 库优化

在 Android 系统中，通过优化和裁剪的 libc 库 Bionic，拥有更高的效率、低内存占用、非常快和小的线程实现、内置了对 Android 特有服务的支持等特点。

Bionic 是 Android 的 C/C++ library，libc 是 GNU/Linux 以及其他类 Unix 系统的基础函数库，最常用的就是 GNU 的 libc，也叫 glibc。Android 之所以采用 bionic 而不是 glibc，有如下 3 个原因：

(1) 版权问题，因为 glibc 是 LGPL；

(2) 库的体积和速度，bionic 要比 glibc 小很多；

(3) 提供了一些 Android 特定的函数，getprop LOGI 等。

Bionic 的主要目录结构及主要功能的说明如下。

```
|-- Android.mk
|-- CleanSpec.mk
|-- libc      (C 库)
|     |-- Android.mk
|     |-- arch-arm   (ARM 构架相关的实现，主要是针对 ARM 的优化，以及和处理器相关的调用)
```

```
|     |-- arch-sh     (ST 公司的 SH4 体系实现)
|     |-- arch-x86    (x86 架构相关的实现)
|     |-- arch-mips (mips 架构相关的实现)
|     |-- bionic
|     |-- CAVEATS
|     |-- docs
|     |-- include
|     |-- inet
|     |-- Jamfile
|     |-- kernel
|     |-- MODULE_LICENSE_BSD
|     |-- netbsd
|     |-- NOTICE
|     |-- private
|     |-- README
|     |-- regex
|     |-- stdio
|     |-- stdlib
|     |-- string
|     |-- SYSCALLS.TXT
|     |-- tools
|     |-- tzcode
|     |-- unistd
|     |-- wchar
|     `-- zoneinfo
|-- libdl         (动态链接库访问接口 dlopen dlsym dlerror dlclose dladdr 的实现)
|     |-- Android.mk
|     |-- arch-sh
|     |-- dltest.c
|     |-- libdl.c
|     |-- MODULE_LICENSE_BSD
|     `-- NOTICE
|-- libm     (C 数学函数库，提供了常见的数序函数和浮点运算)
|     |-- alpha
|     |-- amd64
|     |-- Android.mk
|     |-- arm
|     |-- bsdsrc
|     |-- fpclassify.c
```

```
|       |-- i386
|       |-- i387
|       |-- ia64
|       |-- include
|       |-- isinf.c
|       |-- Makefile-orig
|       |-- man
|       |-- MODULE_LICENSE_BSD_LIKE
|       |-- NOTICE
|       |-- powerpc
|       |-- sh
|       |-- sincos.c
|       |-- sparc64
|       `-- src
|-- libstdc++    (standard c++ lib)
|       |-- Android.mk
|       |-- include
|       |-- MODULE_LICENSE_BSD
|       |-- NOTICE
|       `-- src
|-- libthread_db (线程调试库，可以利用此库对多线程程序进行调试)
|       |-- Android.mk
|       |-- include
|       |-- libthread_db.c
|       |-- MODULE_LICENSE_BSD
|       `-- NOTICE
|-- linker (Android dynamic linker)
|       |-- Android.mk
|       |-- arch
|       |-- ba.c
|       |-- ba.h
|       |-- debugger.c
|       |-- dlfcn.c
|       |-- linker.c
|       |-- linker_debug.h
|       |-- linker_format.c
|       |-- linker_format.h
|       |-- linker.h
|       |-- MODULE_LICENSE_APACHE2
```

```
|      |-- NOTICE
|      |-- README.TXT
|      `-- rt.c
|-- MAINTAINERS
```

另外，在系统移植时也经常用到 bionic。因为本书讲解的 Android 应用开发，所以不介绍相关的内容。

8.4.4 创建进程的优化

Linux 在创建一个新进程时利用了写时拷贝(Copy-on-Write)机制，使得创建一个新的进程非常高效。Android 中每个进程都是基于虚拟机的，并且也要加载基本的库，实际上这些都是可以共享的。基于这方面的考虑，Android 引入了写时拷贝机制，使得 Android 启动一个新的进程，实际上并不消耗很多的内存和 CPU 资源。

另外，Android 在后台一直有个 Zygote 虚拟机在运行，实际上是一个虚拟机实例的孵化器。如果要启动一个新的应用，Zygote 就会创建出一个新的子进程来执行该应用程序，十分高效。图 8-15 显示了 Zygote 创建子进程的过程。

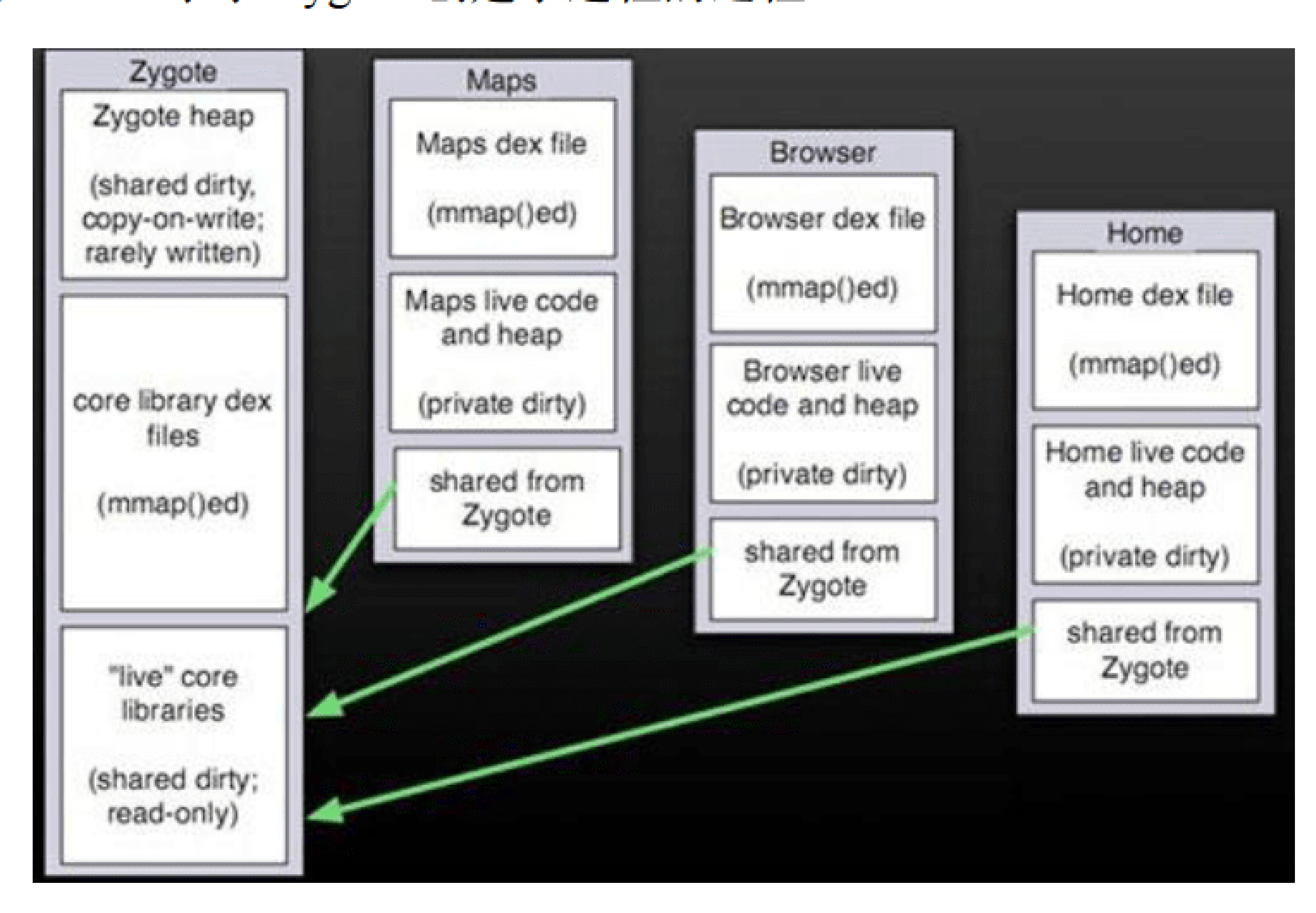


图 8-15 Zygote 创建子进程的过程

当 Android 开机后，会首先启动 Zygote 虚拟机，而不是先启动系统服务器(System Server)，这是出于利用写时拷贝机制创建进程比较高效的考虑。Zygote 虚拟机在启动后会完成虚拟机的初始化、库的加载、预置类库的加载和初始化等操作。并在系统需要一个新的虚拟机对象时，Zygote 可以通过写时拷贝机制高效地创建出新的虚拟机对象。

8.4.5 渲染优化

在进行渲染时，Android 会根据变化的部分进行局部更新，并不是每次都需要重绘整个屏幕。首先计算需要重绘的区域(mInvalidRegion)，如果 DisplayHardware::UPDATE_ON_DEMAND，则通过设定需要重绘的区域的边界来进行局部重绘。

1. Android Browser 快速渲染优化

对于内容繁多复杂的页面来说，当 Browser 进行 scroll 和 zoom 操作时，会显得不流畅，通过阅读 webkit 相关代码发现 webkit 在进行上述操作过程的每一次绘制时，都是整个重绘，没有进行任何缓存操作，所以我们考虑将绘制的页面缓存起来，这样下一次绘制时大部分绘制内容只将上一次的缓存页面进行一个移位(translate)或者缩放(scale)操作，然后再绘制上有内容更新区域上的内容即可。

Android Browser 快速渲染优化所涉及的代码文件如下：

- \frameworks\base\core\java\android\webkit\WebViewCore.java
- \external\webkit\WebKit\android\jni\WebViewCore.h
- \external\webkit\WebKit\android\jni\WebViewCore.cpp

(1) 修改 Java 层

Browser 中的绘制都是调用 WebViewCore.java 中的 drawContentPicture 方法，而 drawContentPicture 又会通过 jni 调用到 webkit 的 C++层。共有三种绘制：normal、zoom、scroll，官方代码是通过对 zoom 绘制和 scroll 绘制 setDrawFilter 的方法进行了小量的优化，我们修改后定义一个 flags 参数区分三种绘制并通过 JNI 调用传到 C++层。

(2) 修改 C++层

在文件 WebViewCore.java 中的，drawContentPicture 最终会调用 webkit 中文件 WebViewCore.cpp 的 drawContent()方法，在该方法中我们调用优化后的绘制实现。 我们在类 WebViewCore 中增加私有成员 int m_contentModSeq 其中前者用来标识不同的绘制请求，每次新的绘制请求都会使该变量加 1。

(3) 实现快速渲染

我们采用双缓存机制，每个缓存对应一个绘制序号，该序号越大表明该缓存越新，具体实现如下：

- normal 绘制：通过 PictureSet 绘制内容到缓存，并从缓存绘制到 canvas。
- zoom 绘制：使用最新的缓存进行 scale 操作，并从缓存绘制到 canvas，不更新缓存。
- scroll 绘制：使用最新的缓存进行 translate 操作，并将其绘制到旧的缓存，通过 PictureSet 绘制更新内容到旧的缓存，最后从该旧的缓存绘制到 canvas，更新旧缓存的绘制序号为最新。

2. 三维场景的渲染优化

对于任何一个 3D 应用程序来说，追求场景画面真实感是一个无止境的目标，其结果就是让我们的场景越来越复杂，模型更加精细，这必然给图形硬件带来极大的负荷以至于无法达到实时绘制帧率。因此，渲染优化是必不可少的。在渲染优化工作之前，我们需要对应用程序性能进行系统的评测，找出瓶颈，然后对症下药。对于 3D 应用程序来说，影响性能的十分多，同时不同的硬件配置条件下，瓶颈也会有所不同。因此，对应用程序进行有效的性能评测，不仅需要对整个渲染管线原理有深入地了解，此外借助一些评测工具能让我们的工作事倍功半。

我们知道渲染流水线的速度是由最慢的阶段决定首先，因此对一个 3D 应用程序进行



评测，首先要分析影响渲染性能的瓶颈是在 CPU 端还是 GPU 端，由此来绝对我们优化的对象。由于目前的图形加速硬件都具有强大的，这个瓶颈往往出现在 CPU 端，我们可以通过一些工具获得这个信息，如 Nvidia 的 NVPerfHUD。在评测选项中，我们可以查看 CPU 和 GPU 繁忙度这项，当 CPU 繁忙度是 100%时，GPU 还不是时，我们知道性能的瓶颈在 CPU 端，我们必须 CPU 端的操作，同时尽量地“喂饱”GPU，把一些费事的计算移植到 GPU 上，例如硬件骨骼蒙皮。当 GPU 端是瓶颈时，说明 GPU 超荷负载，有可能是因为有过多的渲染填充，也就是多边形数量太多(当前强大的 GPU 使得这种情况并不多见。

CPU 上的瓶颈产生有两个方面，一是因为复杂 AI 计算或低效的代码，二是由于不好的渲染批处理或资源管理。对于第一种情况，我们可以利用 VTurn 这类的工具，把应用程序中所有函数调用时间从大到小的排列出来，我们就很容易知道问题所在。对第二种情况来说，同样利用 NVPerfHUD，我们可以查看每帧的 DP 数目，看看批的数量是否过多(有一个具体的换算公式)，查看纹理内存的数目，是否消耗了过多的显存。利用这些工具，我们基本上能够定位应用程序的瓶颈。在应用程序内部，编写一个内嵌的 profiler 功能，能更加便利的进行评测，此外利用 Lua 这样的脚本程序，让我们运行时调试，也能提高评测的效率。

静态场景包括了地形、植被、建筑物等一般不改变位置的实体集合，对它的优化是场景优化中最重主要的内容。在接下来的内容中，将详细讲解静态场景优化的常见问题。

(1) 批的优化

批是场景优化中的最重要的概念之一，它指的是一次渲染调用(DP)，批的尺寸是这次渲染调用所能渲染的多边形数量。每个批的调用都会消耗一定的 CPU 时间，对于显卡来说，一个批里的多边形数量远达不到最大绘制数量。因此尽可能将更多的多边形放在一个批里渲染，以此来减少批的数目，最终降低 CPU 时间，是批的优化基本原则。然而事情往往不尽如人意，有些情况下原有的批会被打破，造成额外的开销，如纹理的改变或不同的矩阵状态。针对这些问题，我们可以采用一些方法来尽量避免它，已达到批尺寸的最大化。

① 合并多个小纹理为一张大纹理。在某个场景中，地面上有十多种不同的植被，它们除了纹理不同外，渲染状态都是一样。我们就可以把它们的纹理打包成一个大纹理，再为每个植被模型指定 UV，这样我们就可以用一个渲染调用来渲染所有的物体，批的数量就从十多个降为一个。这种方法比较适合对纹理精度要求不高，面数不会太多的物体。

② 利用顶点 shader 来统一不同矩阵的情况。即使场景中的所有物体材质都一样，如果它们的矩阵状态不同(特别是场景图管理的引擎)，也会打碎原有的批。利用顶点 shader 技术可以避免这种情况，因为可以把要乘的变换矩阵通过常量寄存器传到 shader 程序中，这样统一了物体的矩阵状态，可以放在一个批里渲染。

(2) 渲染状态管理

渲染状态是用来控制渲染器的渲染行为，在 D3D 中是 setRenderState，通过改变渲染状态，我们可以设置纹理状态、深度写入等等。改变渲染状态对显卡来说，是一个比较耗时的工作，因为显卡执行 API 必须严格按照渲染路径。当渲染状态变化时，显卡就必须执行浮点运算来改变渲染路径，因此给 CPU 和 GPU 带来时间消耗(CPU 必须等待)，渲染状

态变化越大，所要进行的浮点运算越多。因此将渲染状态进行有效的管理，尽可能 减少其变化，对渲染性能影响巨大。(新六代的显卡 Geforce8 系列中将一些常见的状态参数集存储在显卡核心中，当渲染状态发生变化时，可以直接读取保存的参数集，以消除不必要的开销)。绝大部分的 3D 引擎都会按照渲染状态对 PASS 进行分组渲染。

(3) LOD

LOD 这个已经被人讨论烂掉的技术我就不多废话了，简单谈谈一些实际应用。地形的 LOD 我就不多说了，方法太多了，不过感觉目前情况下最实用的还是连锁分片的方法。对于模型 LOD，自动减面的算法，如 VDPM(渐近网格子)并不少见，但是效果都很一般。常规的做法还是让美工做低模进行替换，对于复杂场景来说，模型 LOD 的效果还是比较明显的。材质 LOD 就需要一些技巧，例如可以将雾后的物体，包括地形等统一成一种材质，采用雾的颜色，这样就统一了渲染状态，至于是否要打包成一个 DP 就要看具体情况了(这个统一的材质最好把光照影响关掉，这也是比较费时的)。至于角色模型的 LOD 和普通模型 LOD 相类似，低模减少了顶点数，自然减少了蒙皮计算量。个人认为骨骼 LOD 不是特别的必要，看具体的情况。

(4) 场景管理的优化

场景管理的优化包括场景分割和可见性剔除等。现在的室外场景一般采用 quadtree 或 octree，当我们在性能评测时发现遍历树的过程比较慢时，有可能有两个原因。一是树的深度设置的不合理，我们可以很容易寻找到一个最佳的深度。另一个原因可能是我们为太多数量众多，但体积很小的物体分配了结点，造成结点数量的冗余。解决方法是把这些小物体划分到他们所在的大的结点中。

可见性剔除是最常见优化方法，我们常用的是视锥裁减，这也是非常有效的。视锥裁减也是许多优化方法，这里就不详说了。遮挡裁减也是经常被用到的方法，常见的有地平线裁减。但是在有些情况下，遮挡裁减的效果并不明显，如当 CPU 使用率已经是 100%时，CPU 端是瓶颈，这时进行遮挡裁减计算消耗 CPU 时间，效果就不明显。但是有些情况下利用一些预生成信息的方法，降低遮挡裁减计算的复杂度，提高遮挡裁减计算的效率，对场景性能会一定的改善。

3. 优化浏览器

为页面中所有图片指定宽度和高度，可以消除不必要的 reflows 和重新绘制页面，使页面渲染速度更快。当浏览器勾画页面时，它需要能够流动的，如图片这样的可替换的元素。提供了图片尺寸后，浏览器知道去环绕附近的不可替换元素，甚至可以在图片下载之前开始渲染页面。如果没有指定图片尺寸，或者如果指定的尺寸不符合图片的实际尺寸，一旦图片下载，浏览器将需要回流和重新绘制页面。为了防止 reflows，在 HTML 的<img>标签中或在 CSS 中为所有图片指定宽度和高度。

所以笔者在此提出如下建议：

- 务必指定与图片本身相一致的尺寸。
- 不要使用非图片原始尺寸来缩放图片。如果一个图片文件实际上的大小是 60×60 像素，不要在 HTML 或 CSS 里设置尺寸为 30×30 像素。如果图片需要较小的尺寸，在图像编辑软件中，设置成相一致的尺寸。

- 一定要指定图片或它的块级父元素的尺寸。
- 一定要设置<img>元素本身，或它的块级父元素的尺寸。如果父元素不是块级元素，尺寸将被忽略。不要在一个非最近父元素的祖先元素上设置尺寸。

8.5 SQLite 优化

在 Android 手机应用中，离不开数据库的基本知识，其中最为常用的是 SQLite 数据库。在本节的内容中，将详细讲解优化 SQLite 的基本知识，为读者步入本书后面知识的学习打下基础。

8.5.1 Android SQLite 的查询优化

SQLite 是一个典型的嵌入式 DBMS，它有很多优点，是轻量级的数据库。SQLite 在编译之后很小，其中一个原因就是在查询优化方面比较简单，它只是运用索引机制来进行优化的。

1. 影响查询性能的因素

影响查询性能的因素如下。

- 对表中行的检索数目，越小越好。
- 排序与否。
- 是否要对一个索引。
- 查询语句的形式。

2. 几个查询优化的转换

(1) 单个表的单个列

对于单个表的单个列而言，如果都有形如 T.C=expr 这样的子句，并且都是用 OR 操作符连接起来，形如：

```
x = expr1 OR expr2 = x OR x = expr3
```

此时由于对于 OR，在 SQLite 中不能利用索引来优化，所以可以将它转换成带有 IN 操作符的子句：x IN(expr1,expr2,expr3)这样就可以用索引进行优化，效果很明显，但是如果在都没有索引的情况下 OR 语句执行效率会稍优于 IN 语句的效率。

(2) 一个子句的操作符是 BETWEEN

如果一个子句的操作符是 BETWEEN，在 SQLite 中同样不能用索引进行优化，所以也要进行相应的等价转换，例如：

```
a BETWEEN b AND c
```

可以转换成：

```
(a BETWEEN b AND c) AND (a>=b) AND (a<=c)
```

在上述代码中，(a>=b) AND (a<=c)将被设为 dynamic，并且是(a BETWEEN b AND c)

的子句，那么如果 BETWEEN 语句已经编码，那么子句就忽略不计，如果存在可利用的 index 使得子句已经满足条件，那么父句则被忽略。

(3) 一个单元的操作符是 LIKE

如果一个单元的操作符是 LIKE，那么将做下面的转换：

```
x LIKE 'abc%'
```

转换成：

```
x>='abc' AND x<'abd'
```

因为在 SQLite 中的 LIKE 是不能用索引进行优化的，所以如果存在索引的话，则转换后和不转换相差很远，因为对 LIKE 不起作用，但如果不存在索引，那么 LIKE 在效率方面也还是比不上转换后的效率的。

3. 几种查询语句的处理(复合查询)

(1) 查询语句为：

```
<SelectA> <operator> <selectB> ORDER BY <orderbylist> ORDER BY
```

执行方法为：

```
is one of UNION ALL, UNION, EXCEPT, or INTERSECT
```

这个语句的执行过程是，先将 selectA 和 selectB 执行并且排序，再对两个结果扫描处理，对上面四种操作是不同的，将执行过程分成七个子过程：

- ❑ outA：将 selectA 的结果的一行放到最终结果集中；
- ❑ outB：将 selectA 的结果的一行放到最终结果集中，只有 UNION 操作和 UNION ALL 操作，其它操作都不放入最终结果集中；
- ❑ AltB：当 selectA 的当前记录小于 selectB 的当前记录；
- ❑ AeqB：当 selectA 的当前记录等于 selectB 的当前记录；
- ❑ AgtB：当 selectA 的当前记录大于 selectB 的当前记录；
- ❑ EofA：当 selectA 的结果遍历完；
- ❑ EofB：当 selectB 的结果遍历完。

下面就是 4 种操作的执行过程：

```
UNION ALL
UNION
EXCEPT
INTERSECT
AltB:
outA, nextA
outA, nextA
outA,nextA
nextA
AeqB:
outA, nextA
nextA
nextA
```

```
outA, nextA
AgtB:
outB, nextB
outB, nextB
nextB
nextB
EofA:
outB, nextB
outB, nextB
halt
halt
EofB:
outA, nextA
outA, nextA
outA,nextA
halt
```

(2) 如果可能的话，可以把一个用到 GROUP BY 查询的语句转换成 DISTINCT 语句来查询，因为 GROUP BY 有时候可能会用到 index，而对于 DISTINCT 都不会用到索引的。

4. 子查询扁平化

请看下面的代码：

```
SELECT a FROM (SELECT x+y AS a FROM t1 WHERE z<100) WHERE a>5
```

对这个 SQL 语句的执行，一般默认的方法就是先执行内查询，把结果放到一个临时表中，再对这个表进行外部查询，这就要对数据处理两次，另外这个临时表没有索引，所以对外部查询就不能进行优化了，如果对上面的 SQL 进行处理后可以得到如下 SQL 语句：

```
SELECT x+y AS a FROM t1 WHERE z<100 AND a>5
```

这个结果显然和上面的一样，但此时只需要对数据进行查询一次就够了，另外如果在表 t1 上有索引的话就避免了遍历整个表。

运用 flatten 方法优化 SQL 的条件如下。

(1) 子查询和外查询没有都用集函数。

(2) 子查询没有用集函数或者外查询不是个表的连接。

(3) 子查询不是一个左外连接的右操作数。

(4) 子查询没有用 DISTINCT 或者外查询不是个表的连接。

(5) 子查询没有用 DISTINCT 或者外查询没有用集函数。

(6) 子查询没有用集函数或者外查询没有用关键字 DISTINCT。

(7) 子查询有一个 FROM 语句。

(8) 子查询没有用 LIMIT 或者外查询不是表的连接。

(9) 子查询没有用 LIMIT 或者外查询没有用集函数。

(10) 子查询没有用集函数或者外查询没用 LIMIT。

(11) 子查询和外查询不是同时是 ORDER BY 子句。

(12) 子查询和外查询没有都用 LIMIT。

(13) 子查询没有用 OFFSET。

(14) 外查询不是一个复合查询的一部分或者子查询没有同时用关键字 ORDER BY 和 LIMIT。

(15) 外查询没有用集函数子查询不包含 ORDER BY。

(16) 复合子查询的扁平化：子查询不是一个复合查询，或者他是一个 UNION ALL 复合查询，但他是都由若干个非集函数的查询构成，他的父查询不是一个复合查询的子查询，也没有用集函数或者是 DISTINCT 查询，并且在 FROM 语句中没有其他的表或者子查询，父查询和子查询可能会包含 WHERE 语句，这些都会受到条件(11)、(12)、(13)的限制。

例如下面的 Java 语句：

```
SELECT a+1 FROM (
SELECT x FROM tab
UNION ALL
SELECT y FROM tab
UNION ALL
SELECT abs(z*2) FROM tab2
) WHERE a!=5 ORDER BY 1
```

可以转换为下面的形式：

```
SELECT x+1 FROM tab WHERE x+1!=5
UNION ALL
SELECT y+1 FROM tab WHERE y+1!=5
UNION ALL
SELECT abs(z*2)+1 FROM tab2 WHERE abs(z*2)+1!=5
ORDER BY 1
```

(17) 如果子查询是一个复合查询，那么父查询的所有的 ORDER BY 语句必须是对子查询的列的简单引用。

(18) 子查询没有用 LIMIT 或者外查询不具有 WHERE 语句，子查询扁平化是由专门一个函数实现的，函数为：

```
static int flattenSubquery(
Parse *pParse, /* Parsing context */
Select *p, /* The parent or outer SELECT statement */
int iFrom, /* Index in p->pSrc->a[] of the inner subquery */
int isAgg, /* True if outer SELECT uses aggregate functions */
int subqueryIsAgg /* True if the subquery uses aggregate functions */
)
```

它是在文件 Select.c 中实现的，显然这对于一个比较复杂的查询，如果满足上面的条件时对这个查询语句进行扁平化处理后就可以实现对查询的优化，如果正好存在索引的话效果会更好。

5. 连接查询

在返回查询结果之前，相关表的每行必须都已经连接起来，在 SQLite 中，这是用嵌套循环实现的，在早期版本中，最左边的是最外层循环，最右边的是最内层循环，连接两个或者更多的表时，如果有索引则放到内层循环中，也就是放到 FROM 最后面，因为对于前面选中的每行，找后面与之对应的行时，如果有索引则会很快，如果没有则要遍历整个

表，这样效率就很低，但在新版本中，这个优化已经实现。

具体优化的方法如下。

对要查询的每个表，统计这个表上的索引信息，首先将代价赋值为 SQLITE_BIG_DBL(一个系统已经定义的常量)：

(1) 如果没有索引，则找有没有在这个表上对 rowid 的查询条件：

- 如果有 Rowid=EXPR，如果有的话则返回对这个表代价估计，代价计为零，查询得到的记录数为 1，并完成对这个表的代价估计。
- 如果没有 Rowid=EXPR 但有 rowid IN(...)，而 IN 是一个列表，那么记录返回记录数为 IN 列表中元素的个数，估计代价为 NlogN。
- 如果 IN 不是一个列表而是一个子查询结果，那么由于具体这个子查询不能确定，所以只能估计一个值，返回记录数为 100，代价为 200。
- 如果对 rowid 是范围的查询，那么就估计所有符合条件的记录是总记录的三分之一，总记录估计为 1000000，并且估计代价也为记录数。
- 如果这个查询还要求排序，则再另外加上排序的代价 NlogN。
- 如果此时得到的代价小于总代价，那么就更新总代价，否则不更新。

(2) 如果 WHERE 子句中存在 OR 操作符，那么要把这些 OR 连接的所有子句分开再进行分析。

- 如果有子句是由 AND 连接符构成，那么再把由 AND 连接的子句再分别分析。
- 如果连接的子句的形式是 X<op><expr>，那么就再分析这个子句。
- 接下来就是把整个对 OR 操作的总代价计算出来。
- 如果这个查询要求排序，则再在上面总代价上再乘上排序代价 NlogN。
- 如果此时得到的代价小于总代价，那么就更新总代价，否则不更新。

(3) 如果有索引，则统计每个表的索引信息，对于每个索引：

先找到这个索引对应的列号，再找到对应的能用到(操作符必须为=或者是 IN(…))这个索引的 WHERE 子句，如果没有找到，则退出对每个索引的循环，如果找到，则判断这个子句的操作符是什么，如果是=，那么没有附加的代价，如果是 IN(sub-select)，那么估计它附加代价 inMultiplier 为 25，如果是 IN(list)，那么附加代价就是 N(N 为 list 的列数)。

再计算总的代价和总的查询结果记录数和代价。

```
nRow = pProbe->aiRowEst * inMultiplier;/*计算行数*/
cost = nRow * estLog(inMultiplier);/*统计代价*/
```

如果找不到操作符为=或者是 IN(…)的子句，而是范围的查询，那么同样只好估计查询结果记录数为 nRow/3，估计代价为 cost/3。

同样，如果此查询要求排序的话，再在上面的总代价上加上 NlogN。如果此时得到的代价小于总代价，那么就更新总代价，否则不更新。

(4) 通过上面的优化过程，可以得到对一个表查询的总代价。

再对第二个表进行同样的操作，这样如此直到把 FROM 子句中所有的表都计算出各自的代价，最后取最小的，这将作为嵌套循环的最内层，依次可以得到整个嵌套循环的嵌套顺序，此时正是最优的，达到了优化的目的。

(5) 所以循环的嵌套顺序不一定是与 FROM 子句中的顺序一致，因为在执行过程中会

用索引优化来重新排列顺序。

6. 索引

在 SQLite 中，有以下 4 种索引。

- 单列索引。
- 多列索引。
- 唯一性索引。
- 对于声明为 INTEGER PRIMARY KEY 的主键来说，这列会按默认方式排序，所以虽然在数据字典中没有对它生成索引，但它的功能就像个索引。所以如果在这个主键上在单独建立索引的话，这样既浪费空间也没有任何好处。

在运用索引时，需要注意如下三条事项：

- 对于一个很小的表来说没必要建立索引。
- 在一个表上如果经常做的是插入更新操作，那么就要节制使用索引。
- 也不要在一个表上建立太多的索引，如果建立太多的话那么在查询的时候 SQLite 可能不会选择最好的来执行查询，一个解决办法就是建立聚簇索引。

在如下情况下运用索引：

(1) 操作符：=、>、<、IN 等。

(2) 操作符 BETWEEN、LIKE、OR 不能用索引，例如：

```
BETWEEN: SELECT * FROM mytable WHERE myfield BETWEEN 10 and 20;
```

这时就应该将其转换成：

```
SELECT * FROM mytable WHERE myfield >= 10 AND myfield <= 20;
```

此时如果在 myfield 上有索引的话就可以用了，大大提高速度。再如：

```
LIKE: SELECT * FROM mytable WHERE myfield LIKE 'sql%';
```

此时应该将它转换成：

```
SELECT * FROM mytable WHERE myfield >= 'sql' AND myfield < 'sqm';
```

此时如果在 myfield 上有索引的话就可以用了，大大提高了速度。再如 OR：

```
SELECT * FROM mytable WHERE myfield = 'abc' OR myfield = 'xyz';
```

此时应该将它转换成：

```
SELECT * FROM mytable WHERE myfield IN ('abc', 'xyz');
```

此时如果在 myfield 上有索引的话就可以用了，大大提高速度。

有些时候索引都是不能用的，这时就应该遍历全表，例如下面的语句：

```
SELECT * FROM mytable WHERE myfield % 2 = 1;
SELECT * FROM mytable WHERE substr(myfield, 0, 1) = 'w';
SELECT * FROM mytable WHERE length(myfield) < 5;
```

8.5.2 SQLite 性能优化技巧

由于支持 SQL 语言查询底层开源整体性能表现得比较稳定，我们可以通过三点提高 Android 数据库性能。

(1) 相对于封装过的 ContentProvider 而言，使用原始 SQL 语句执行效率高，比如使用方法 rawQuery、execSQL 的执行效率比较高。

(2) 对于需要一次性修改多个数据时，可以考虑使用 SQLite 的事务方式批量处理，我们定义 SQLiteDatabase db 对象，执行的顺序为：

```
db.beginTransaction();
//这里处理数据添加，删除或修改的 SQL 语句
db.setTransactionSuccessful();                //在此设置处理成功
//告诉数据库处理完成了，这时 SQLite 的底层会执行具体的数据操作
db.endTransaction();
```

(3) 打好 SQL 语句的基础，对于查询，以及分配表的结构都十分重要，建议有时间的网友可以查看 SQLite 的源码，了解底层的具体实现，加深了解后可以很好地实现性能调优。

8.6 Android 的图片缓存处理和性能优化

在本节下面的内容中，将简要介绍 Android 的图片缓存处理和性能优化的基本知识。

(1) 在使用 Gallery 控件时，如果载入的图片过多、过大，就很容易出现 OutOfMemoryError 异常，就是内存溢出。这是因为 Android 默认分配的内存只有几 M，而载入的图片如果是 JPG 之类的压缩格式，在内存中展开时就会占用大量的空间，也就容易内存溢出。这时可以用下面的方法解决：

```
ImageView i = new ImageView(mContext);
         BitmapFactory.Options options=new BitmapFactory.Options();
         options.inSampleSize = 10;
         //貌似这个 options 的功能是返回缩略图，10 即表示长和宽为原来的 1 / 10，即面积
为原来的 1 / 100
         //缩略图可以减少内存占用
         Bitmap bm = BitmapFactory.decodeFile(lis.
                          get(position).toString(),options);
         i.setImageBitmap(bm);
         bm.recycle();
         //资源回收
```

(2) 实现统一管理位图资源，适时释放资源的演示代码如下。

```
class ImageManager {
   private WeakHashMap<Integer, WeakReference<Bitmap>> mBitmaps;
   private WeakHashMap<Integer, WeakReference<Drawable>> mDrawables;
```

```
    private boolean mActive = true;

    public ImageManager() {
        mBitmaps = new WeakHashMap<Integer, WeakReference<Bitmap>>();
        mDrawables = new WeakHashMap<Integer, WeakReference<Drawable>>();
    }


    public Bitmap getBitmap(int resource) {
        if (mActive) {
            if (!mBitmaps.containsKey(resource)) {
                mBitmaps.put(resource,
                    new WeakReference<Bitmap>(BitmapFactory.decodeResource
(MainActivity.getContext().getResources(), resource)));
            }
            return ((WeakReference<Bitmap>)mBitmaps.get(resource)).get();
        }
        return null;
    }

    public Drawable getDrawable(int resource) {
        if (mActive) {
            if (!mDrawables.containsKey(resource)) {
                mDrawables.put(resource, new WeakReference<Drawable>
(getApplication().getResources().getDrawable(resource)));
            }
            return ((WeakReference<Drawable>)mDrawables.get(resource)).get();
        }
        return null;
    }
    public void recycleBitmaps() {
        Iterator itr = mBitmaps.entrySet().iterator();
        while (itr.hasNext()) {
            Map.Entry e = (Map.Entry)itr.next();
            ((WeakReference<Bitmap>) e.getValue()).get().recycle();
        }
        mBitmaps.clear();
    }
    public ImageManager setActive(boolean b) {
        mActive = b;
        return this;
    }
    public boolean isActive() {
        return mActive;
    }
}
```

(3) 网络连接往往是耗电量比较大的，我们可以优化一下在需要网络连接的程序中，首先检查网络连接是否正常，如果没有网络连接，那么就不需要执行相应的程序。检查网络连接的演示代码如下：



```
private boolean isConnected(){
        ConnectivityManager mConnectivity = (ConnectivityManager)
this.getSystemService(CONNECTIVITY SERVICE);
        TelephonyManager mTelephony =
(TelephonyManager)getSystemService(Context.TELEPHONY SERVICE);

        // 检查网络连接，如果无网络可用，就不需要进行联网操作等
        NetworkInfo info = mConnectivity.getActiveNetworkInfo();
        if (info == null ||
               !mConnectivity.getBackgroundDataSetting()) {
               return false;
        }
        //判断网络连接类型，只有在 3G 或 wifi 里进行一些数据更新。
        int netType = info.getType();
        int netSubtype = info.getSubtype();
        if (netType == ConnectivityManager.TYPE WIFI) {
            return info.isConnected();
        } else if (netType == ConnectivityManager.TYPE MOBILE
               && netSubtype == TelephonyManager.NETWORK TYPE UMTS
               && !mTelephony.isNetworkRoaming()) {
           return info.isConnected();
        } else {
           return false;
        }
    }
```

(4) 网络间的数据传输也是非常耗费资源的，这包括传输方式和解析方式，如图 8-16 所示的统计图。

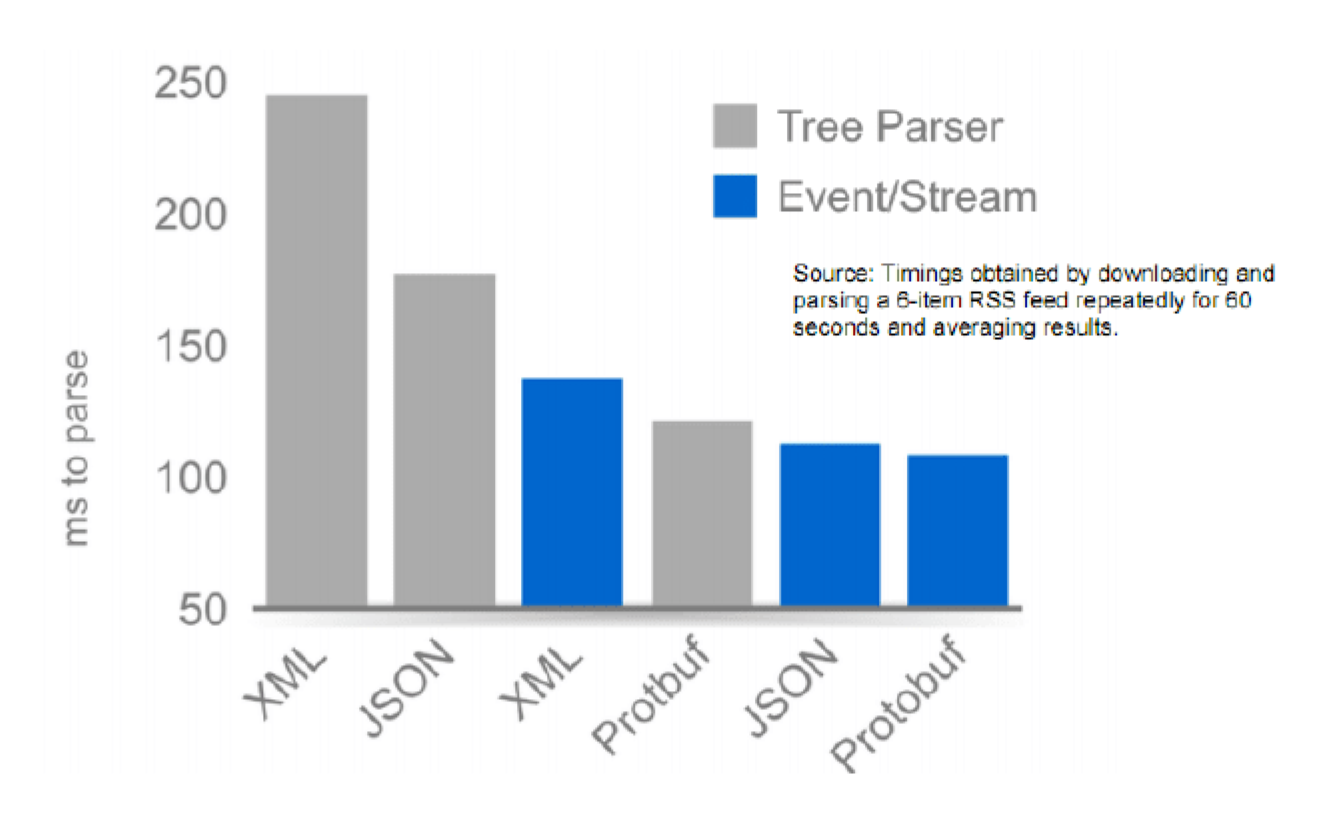


图 8-16　统计图

其中 Tree Parse 是 DOM 解析 Event/Stream 是 SAX 方式解析。很明显，使用流的方式解析效率要高一些，因为 DOM 解析是在对整个文档读取完后，再根据节点层次等再组织起来。而流的方式是边读取数据边解析，数据读取完后，解析也就完毕了。

而在数据格式方面，JSON 和 Protobuf 效率明显比 XML 好很多，XML 和 JSON 大家都很熟悉。从图 8-8 中我们可以得出结论就是尽量使用 SAX 等边读取边解析的方式来解析数据，针对移动设备，最好能使用 JSON 之类的轻量级数据格式为佳。

(5) 传输数据经过压缩目前大部门网站都支持 GZIP 压缩，所以在进行大数据量下载时，尽量使用 GZIP 方式下载，可以减少网络流量。演示代码如下。

```
HttpGet request = new HttpGet("http://example.com/gzipcontent");
    HttpResponse resp =
       new DefaultHttpClient().execute(request);
    HttpEntity entity = response.getEntity();
    InputStream compressed = entity.getContent();
    InputStream rawData = new GZIPInputStream(compressed);
```

(6) 有效管理 Service 后台服务就相当于一个持续运行的 Acitivity。如果开发的程序后台都有一个 service 不停地去服务器上更新数据，在不更新数据的时候就让它 sleep，这种方式是非常耗电的。通常情况下，我们可以使用 AlarmManager 来定时启动服务，每 30 分钟执行一次。演示代码如下。

```
AlarmManager am =
(AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
        Intent intent = new Intent(context, MyService.class);
        PendingIntent pendingIntent = PendingIntent.getService(context,
0, intent, 0);
        long interval = DateUtils.MINUTE_IN_MILLIS * 30;
        long firstWake = System.currentTimeMillis() + interval;
        am.setRepeating(AlarmManager.RTC,firstWake, interval,
pendingIntent);
```

在此建议读者，在开发过程中应该注意一些细节，并且手机的整体性能和续航都有很大的局限性，很多优化的细节都会对软件产生本质的影响，这要引起重视。

Android

第9章 系统优化

系统优化原来是系统科学(系统论)的术语，现在也用作(而且常用作)计算机方面的术语。系统优化的目标是，尽可能地减少计算机执行少的进程，更改工作模式，删除不必要的中断，让机器运行更有效，优化文件位置使数据读写更快，空出更多的系统资源供用户支配，以及减少不必要的系统加载项及自启动项。当然优化到一定程度可能会略微影响系统稳定性，但基本对硬件无害。在本章的内容中，将详细讲解 Android 系统优化的基本知识，为读者步入本书后面高级知识的学习打下基础。

9.1 基本系统优化

目前智能手机市场主要有两大系统，分别是 iOS 和 Android，并且已经形成了鲜明的对立阵营。在 iOS 用户眼中，Android 的形象几乎可以用一个“卡”字来代替。其实 Android 用户无须理会 iOS 究竟有多么好的用户体验，目前 Android 经过了版本的升级和发展，硬件水平已经有了很大的提高，再加上目前软件系统自身的优化，Android“卡”的情况已经有了很大程度的缓解。目前的双核机型硬件配置十分强大，如果还要说“卡”，也就是因为厂商定制 ROM 的优化原因。其实 Android 的“卡”，可以得到彻底地解决，这就关系到了 Android 的优化问题。

9.1.1 刷机重启

这是 Android 用户的一大乐趣，部分用户刷机是为了得到更好的易用性，比如小米的 MIUI ROM，非常符合中国人的使用习惯，也有着足够丰富的个性化设定，是图省事的朋友的最好选择之一。不过对于追求高性能的朋友来说，MIUI 的优化还有很大提升空间，人们纷纷选择了对于 ROM 优化更加出色的 CyanogenMod 作为刷机的第一选择。

CyanogenMod 系列目前主打的 ROM 有 CM 7.2 和 CM 9 两个，CM 7.2 基于 Android 2.3.7，而 CM 9 则基于 Android 4.0.4，其中 CM 7.2 已经基本成熟，完美支持的机型很多，是大部分机友刷机的第一选择，CM 9 官方的 ROM 支持机型并不多，民间高手也都进行了各个机型的移植，官方支持的机型兼容性相当不错，而移植情况并不乐观。

CM 系列 ROM 忠实于 AOSP，在底层驱动方面做了很多努力，刷入之后就会感觉手机流畅了许多，同时也支持了更多的美化和手机自定义能力，比如我们可以对手机的震动回馈做细致的调整，包括按下震动的强度，抬起震动的强度等，让手机虚拟按键给我们更为真实的回馈，在 CM ROM 中，类似的设定非常多。

目前大部分的 ROM 都是使用 CM 进行定制的，还有一部分是对官方原版 ROM 进行修改，仅有少部分的 ROM 是修改的 AOSP 的源码，这些 ROM 指向都是谷歌 Nexus 系列的机型，比如 GALAXY Nexus 和 Nexus S 上的 Codename 和 AOKP，就针对源码做了很多修改，让手机变得更流畅。

9.1.2 刷内核

仅仅刷手机的 ROM 是不够的，虽然多了很多自定义的功能，流畅度已经高于官方的 ROM，但是依旧有很大的提升空间。这时候我们就需要通过刷内核来进一步优化，刷内核所能带来的提升是相当明显的，但是对于刷内核大家还是要谨慎。

刷内核相比刷 ROM，是一个很小的工程，我们的手机不必具备 Wipe，也就是说不用删除手机内部的数据，刷一下也就几分钟的工夫。所以刷内核的时候，大家完全可以多下几个内核，逐个进行测试，看看哪个内核更适合自己，就保留哪个内核。同时刷内核时我们要注意，内核需对应自己的手机版本，对应自己所刷的 ROM，否则会造成手机无法启


动的现象，如果遇到无法启动的现象，再刷其他可用内核就可以恢复。

究竟刷内核到有什么用呢？首先是超频，大部分内核会默认提供降压超频，并拥有多种超频策略，来保证超频的情况下更省电。其次，还提供更多调整，比如内存虚拟机的大小、颜色管理等，甚至一个内核可以包括一些新的 Linux 的补丁，比如最新的 Linux 3.3 所集成的 CPU 频率补丁等。

事实上，一般的第三方 ROM 已经修改了手机的内核，达到了更流畅的目的，但 ROM 的制作速度远远比不上内核的调整速度，有时候一个 ROM 适用的内核在一天之内可能多次更新，所以我们可以尝试不同的新内核，看看他们的超频是不是能给我们带来性能上质的提升，是不是能更省电，是不是能通过颜色调整让我们看到更棒的画面等。

9.1.3　精简内置应用

经过 9.1.1 节和 9.1.2 节的学习，相信 Android 用户通过不断的更换 ROM 和刷内核已经在流畅度上有了质的飞跃了。如果还是不满意，我们还有其他的路可选，接下来将要讲解的精简内置应用就是一个可以大幅度提升流畅度的方法。例如 Google 提供的服务就是大部分人精简的对象。

Android 系统和 iOS、Windows Phone 系统不同，Android 系统拥有真正的后台运行能力。虽然 iOS 在推送方面做得很好，弥补了后台方面的不足，但是仍然无法与 Android 的真后台相比，但是由于 Android 的程序优先级并不像 iOS 和 Windows Phone 那样，为了流畅让当前界面拥有最高优先级，所以需要关掉 Android 手机后台中不必要的进程，专业可以获得最佳的性能。精简界面如图 9-1 所示。



图 9-1　精简界面效果

此时精简内部应用就是很好的选择，因为在我们的使用过程中，有许多 Android 内部应用程序是不必要的，而且这些程序会在我们不用的时候悄悄地打开后台，对我们的使用造成影响。在精简时需要用到 root 文件管理器，同时需要保证手机已经开启 root 权限，如图 9-2 所示。

这时进入“system/app”就可以进行精简操作了，我们需要把 root 管理器的当前权限

设置成读写，并且修改需要删除的软件权限，打开软件执行操作的权限，就可以删除内置软件了。在精简操作前，需要对软件进行备份，或者备份整个ROM。如果不慎精简掉系统程序，可能会造成无法开机的情况，这时需要重刷 ROM 解决。所以在此建议读者：最好找到该机型、该 ROM 的精简列表，以避免重复劳动。



图 9-2　root 文件管理器界面

9.1.4　基本系统优化总结

对于 Android 系统来说，流畅度是它相比 iOS 系统最大的短板。其实 Android 的大部分手机有着相当好的硬件，所以流畅度大幅度提升完全不是难事，而各个厂商在 Android 手机出厂前给手机定制的 ROM 并没有达到最优的优化效果，或多或少都有可提升的空间。

所以读者们可以根据自己用手机的需要对手机进行彻头彻尾的优化，从 ROM 开始让手机变得彻底流畅起来。在此需要提醒大家注意如下两点。

(1) 一定要选择普及率较高的 Android 机型，尤其是在国外的高普及度，像谷歌的 Nexus 系列手机被誉为亲儿子，也是因为它开放了源代码，在其他手机为第三方 ROM 挠头的时候，Nexus 系列已经早早地开始各种优化了。

(2) 除了 ROM 资源，我们也要考虑其他资源，比如内核、各大手机厂商的热门机型、内核资源也是不一样的，早期摩托罗拉的里程碑很开放，所以有着大量可刷的内核，而到了后来摩托罗拉机型很封闭，可刷的内核资源就相当匮乏，虽然 ROM 很多，但刷来刷去都大同小异，刷机的乐趣锐减。这里谷歌的 Nexus 系列再一次做了表率。另外，Android 的官方站点提供的资源也可以拿来使用，如图 9-3 所示。

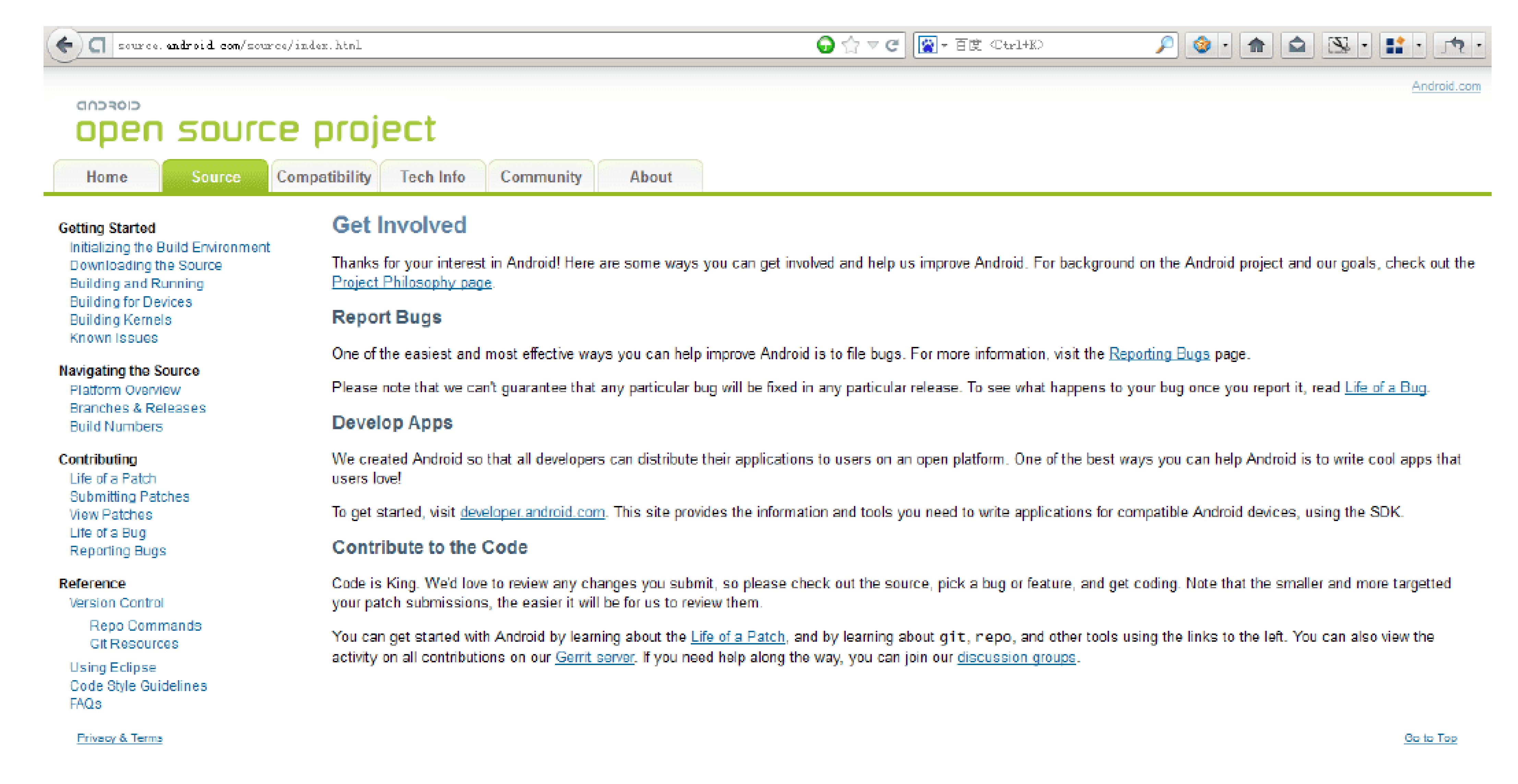


图 9-3　Android 开发资源界面

9.2 进 程 管 理

在本书前面的内容中，曾经讲解过关掉进程不一定是一件好事。这是因为在 Android 里，进程和程序是两回事，程序可以一直保留在系统里，但是没有任何进程在后台“运行”，也不消耗任何系统资源。所有的程序保留在内存中，所有可以更快地启动回到它之前的状态。当你的内存用完了，系统会自动帮你杀掉你不用的任务。

9.2.1 Android 进程跟 Windows 进程是两回事

我们需要明白的是，Android 用 RAM 的方式，跟 Windows 是两回事。在 Android 的世界里面，RAM 被用满了是件“好”事。它意味着可以快速打开之前打开的软件，回到之前的位置。所以 Android 很有效的使用 RAM，很多用户看到他们的 RAM 满了，就认为拖慢了他们的电话。而实际上，是我们的 CPU，当软件真正运行时用到的才是拖慢手机的瓶颈。

很流行的各种进程管理软件都说帮你释放内存是件好事，但这是不正确的。打开这些软件时，它们告诉你“运行”的软件和杀死他们的方法。你也可以在“服务”里面看到到底程序的哪些部分在“运行”，占用了多少内存，剩余多少内存。所有的这些都告诉你，杀掉这些程序能够释放内存。但是这些软件都没有告诉你这些程序到底消耗了多少 CPU 时钟，而仅仅告诉你能释放多少内存。要知道，用满了内存实际上是件好事，我们要注意的是 CPU，真正消耗你的手机资源，消耗电池。

因此，杀掉程序通常是没有必要的(尤其是用 autokill 方式杀掉程序)。更严重的是，这样做会更快地拖垮你的手机能力和电池性能。不管是手动杀掉进程，还是自动地杀掉进程，重新打开程序，实际上是在用 CPU 资源来做这件事。

事实上，这些进程管理软件消耗了系统资源。而且，这些软件会莫名其妙地杀死其他程序造成乱七八糟的结果(尤其对些小白来说)。由此可以看出，用这些进程管理软件耽误的事情比得到的要多。也可以由此看出，国内用户和开发者被 Windows 都给教坏了，在 Windows 上的一些习惯也被都带到了手机上面。这么晚出现的一个平台，在进程管理上不可能赶不上一个 Task Killer。

9.2.2 查看当前系统中正在运行的程序

尽管关闭进程的缺点大于优点，但是身为程序员，还是需要掌握开发进程应用相关的知识。在接下来的内容中，先讲解开发一个查看当前系统中正在运行程序的实现过程。

实例 1	
源码路径	\daima\9\jincheng
功能	查看当前系统中正在运行的程序

在本实例中插入了一个按钮，单击按钮后会显示当前系统中正在运行的程序。当前运行程序是通过 ActivityManager.getRunningTasks 方法获取的，然后通过 ListView 将获取的信息显示出来。当单击按钮后，如果在 ListView 的工作已经结束或被操作系统回收，则是不会更新运行列表的。另外，如果不具有访问其他运行程序的权限，也不会显示在 ListView 列表中。为了保证 Android 的运行，限制了获取于小程序的数量，在本实例中设置了最多获取 30 个进程。

本实例的具体实现流程如下。

(1) 编写的文件是 example.java，其具体实现流程如下。

- 设置类成员最多能够获取 30 个 Task 数量，具体代码如下。

```
/* 类成员设置取回最多几笔的 Task 数量 */
private int intGetTastCounter=30;
```

- 设置类成员 ActivityManager 的对象，当单击按钮后取得正在后台运行的工作程序，具体代码如下。

```
/* 类成员 ActivityManager 对象 */
private ActivityManager mActivityManager;
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  mButton01 = (Button)findViewById(R.id.myButton1);
  mListView01 = (ListView)findViewById(R.id.myListView1);

  /* 单击按钮取得正在后台运行的工作程序 */
  mButton01.setOnClickListener(new Button.OnClickListener()
  {
    @Override
    public void onClick(View v)
    {
      // TODO Auto-generated method stub
      try
      {
        /* ActivityManager 对象向系统取得 ACTIVITY_SERVICE */
        mActivityManager = (ActivityManager)
        example.this.getSystemService(ACTIVITY_SERVICE);

        arylistTask = new ArrayList<String>();

        /* 以 getRunningTasks 方法取回正在运行中的程序 TaskInfo */
        List<ActivityManager.RunningTaskInfo> mRunningTasks =
        mActivityManager.getRunningTasks(intGetTastCounter);

        int i = 1;
```



```
        /* 以循环及 baseActivity 方式取得工作名称与 ID */
        for (ActivityManager.RunningTaskInfo amTask : mRunningTasks)
        {
          /* baseActivity.getClassName 取出运行工作名称 */
          arylistTask.add("" + (i++) + ": "+
          amTask.baseActivity.getClassName()+
          "(ID=" + amTask.id +")");
        }
        aryAdapter1 = new ArrayAdapter<String>
        (example17.this, R.layout.simple_list_item_1, arylistTask);

        if(aryAdapter1.getCount()==0)
        {
          /* 当没有任何运行的工作，则提示信息 */
          mMakeTextToast
          (
            getResources().getText
            (R.string.str_err_no_running_task).toString(),
            true
          );
        }
        else
        {
          /* 发现后台运行的工作程序，以 ListView Widget 条列呈现 */
          mListView01.setAdapter(aryAdapter1);
        }
      }
      catch(SecurityException e)
      {
        /* 当无 GET_TASKS 权限时(SecurityException 异常)提示信息 */
        mMakeTextToast
        (
          getResources().getText
          (R.string.str_err_permission).toString(),
          true
        );
      }
    }
  });
```

❑ 监听用户选择某一个正在运行进程时的事件，具体代码如下。

```
  mListView01.setOnItemSelectedListener
  (new ListView.OnItemSelectedListener()
  {
    @Override
    public void onItemSelected
    (AdapterView<?> parent, View v, int id, long arg3)
    {
      // TODO Auto-generated method stub
      /* 由于将运行工作以数组存放，所以使用 id 取出数组元素名称 */
```

```
        mMakeTextToast(arylistTask.get(id).toString(),false);
      }
      @Override
      public void onNothingSelected(AdapterView<?> arg0)
      {
        // TODO Auto-generated method stub

      }
    });
```

- 设置当用户选择某一个正在运行进程时的事件处理，具体代码如下。

```
    /* 当 User 在运行工作上点击时的事件处理 */
    mListView01.setOnItemClickListener
    (new ListView.OnItemClickListener()
    {
      @Override
      public void onItemClick
      (AdapterView<?> parent, View v, int id,  long arg3)
      {
        // TODO Auto-generated method stub
        /* 由于将运行工作以数组存放，故以 id 取出数组元素名称 */
        mMakeTextToast(arylistTask.get(id).toString(), false);
      }
    });
  }
```

- 定义方法 mMakeTextToast(String str, boolean isLong)实现一个提醒效果，具体代码如下。

```
  public void mMakeTextToast(String str, boolean isLong)
  {
    if(isLong==true)
    {
      Toast.makeText(example17.this, str, Toast.LENGTH_LONG).show();
    }
    else
    {
      Toast.makeText(example17.this, str, Toast.LENGTH_SHORT).show();
    }
  }
}
```

(2) 编写文件 AndroidManifest.xml，在此文件中声明 GET_TASKS 权限，具体代码如下。

```
<uses-permission android:name="android.permission.GET_TASKS">
</uses-permission>
```

执行后在屏幕中显示一个按钮，如图 9-4 所示。单击【获取运行的程序】按钮后列表显示当前正在运行的程序，如图 9-5 所示。

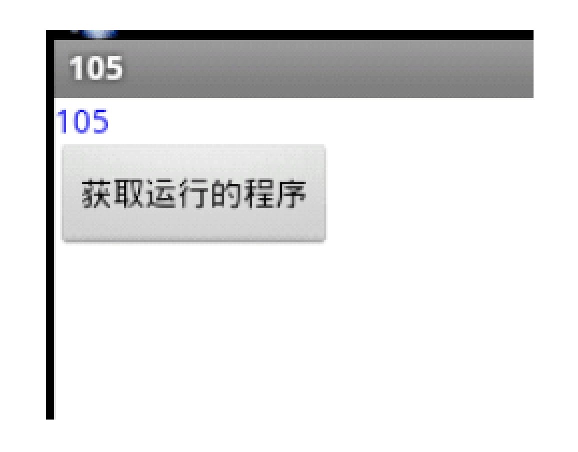


图 9-4　初始效果



图 9-5　当前运行程序

9.2.3　枚举 Android 系统的进程、任务和服务的信息

在接下来的内容中，将详细讲解枚举 Android 系统的进程、任务和服务的信息的方法。最终目的是返回当前正在运行的任务列表(任务是一个或多个活动的集合，这些活动以栈的形式运行在一个任务当中)，按照最近一次运行的任务排在任务列表前端的方式，输出所有的任务。

假设我们的预期目标是：使用三个 Tab 页来分别显示进程信息、任务信息和服务信息，每个 Tab 页中都是一个 ListActivity，以列表的方式进行展示。预期效果如图 9-6～图 9-8 所示。

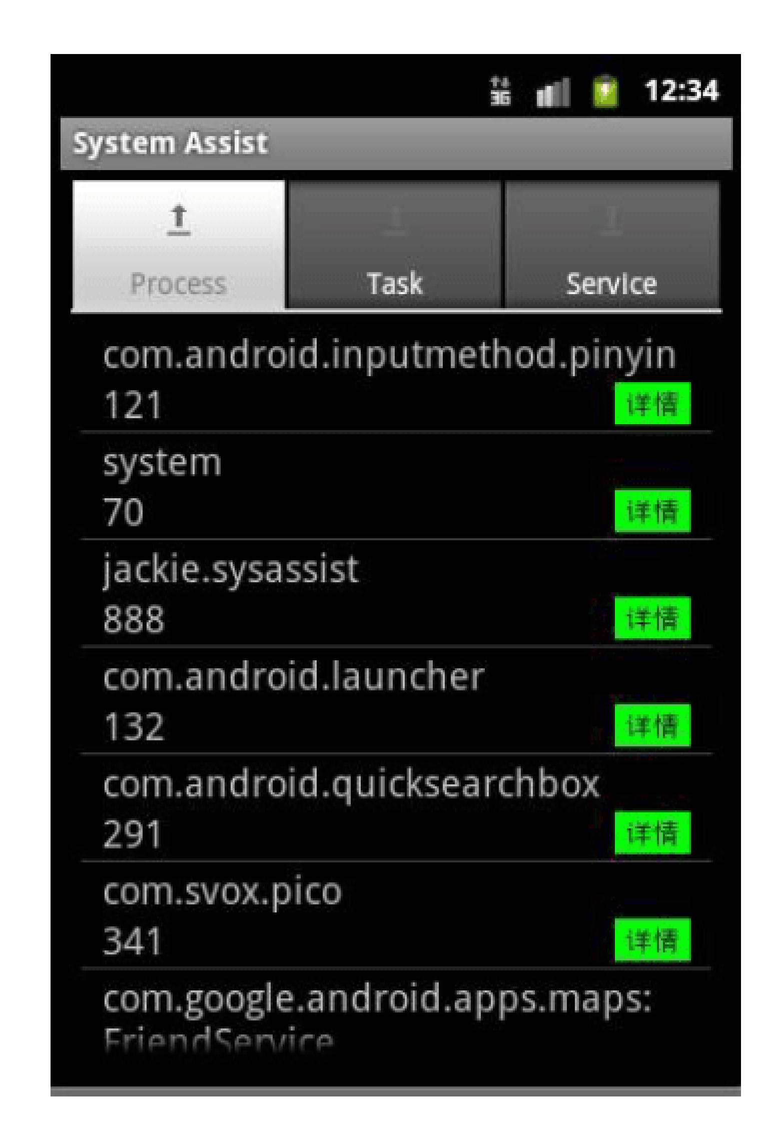


图 9-6　预期系统进程信息效果

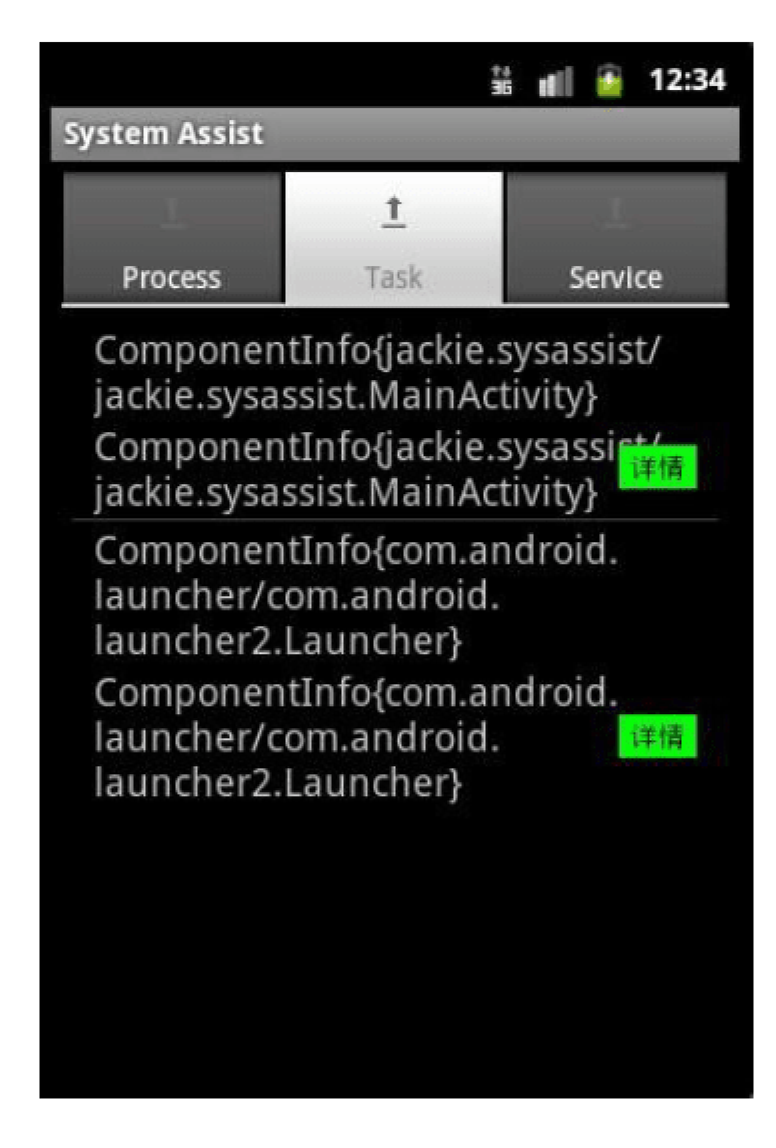


图 9-7　预期系统任务信息效果





图 9-8　预期系统服务信息

根据上述预期效果和功能，接下来开始演示具体实现过程。

(1) 首先获取 ActivityManager 的对象实例，通过调用 getSystemService (ACTIVITY_SERVICE)返回一个 ActivityManager 的实例。在获取该实例后，调用其 getRunningAppProcesses()方法返回一个 List，在该 List 中存放的数据类型为 ActivityManager.RunningAppProcessInfo。然后对该 List 进行遍历，从 List 中的每项 RunningAppProcessInfo 中可以获取尽享相关的信息。例如在下面的演示代码中，使用了一个 ListAdapter 来绑定到一个 ListView 中进行显示。

```
/**
* ActivityManager.RunningAppProcessInfo {
*     public int importance                  // 进程在系统中的重要级别
*     public int importanceReasonCode         // 进程的重要原因代码
*     public ComponentName importanceReasonComponent  // 进程中组件的描述信息
*     public int importanceReasonPid          // 当前进程的子进程 Id
*     public int lru                         // 在同一个重要级别内的附加排序值
*     public int pid                         // 当前进程 Id
*     public String[] pkgList                 // 被载入当前进程的所有包名
*     public String processName              // 当前进程的名称
*     public int uid                         // 当前进程的用户 Id
* }
*/
package crazypebble.sysassist.procmgr;
import crazypebble.sysassist.R;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
```

```
    import android.app.ActivityManager;
    import android.app.ActivityManager.RunningAppProcessInfo;
    import android.app.ListActivity;
    import android.os.Bundle;
    import android.widget.SimpleAdapter;

    public class ProcMgrActivity extends ListActivity {

        private static List<RunningAppProcessInfo> procList = null;

        @Override
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.proc list);

            procList = new ArrayList<RunningAppProcessInfo>();
            getProcessInfo();

            showProcessInfo();
        }

        public void showProcessInfo() {

            // 更新进程列表
            List<HashMap<String,String>> infoList = new
ArrayList<HashMap<String,String>>();
            for (Iterator<RunningAppProcessInfo> iterator =
procList.iterator(); iterator.hasNext();) {
                RunningAppProcessInfo procInfo = iterator.next();
                HashMap<String, String> map = new HashMap<String, String>();
                map.put("proc name", procInfo.processName);
                map.put("proc id", procInfo.pid+"");
                infoList.add(map);
            }

            SimpleAdapter simpleAdapter = new SimpleAdapter(
                    this,
                    infoList,
                    R.layout.proc list item,
                    new String[]{"proc name", "proc id"},
                    new int[]{R.id.proc name, R.id.proc id} );
            setListAdapter(simpleAdapter);
        }

        public int getProcessInfo() {
            ActivityManager activityManager = (ActivityManager)
getSystemService(ACTIVITY SERVICE);
            procList = activityManager.getRunningAppProcesses();
            return procList.size();
        }
    }
```

(2) 开始获取系统任务的信息。获取系统的任务信息的方法跟获取进程的方法差不多，只不过在得到 ActivityManager 的实例之后，调用的是 getRunningTasks(maxTaskNum)方法，参数 maxTaskNum 限定了所要获取的最大的任务数目，如果系统中的任务总数比这个数值小，我们可以得到系统所有的任务信息；但是如果系统的任务总数比这个参数的值要大的话，就只能获得该值所限定的任务个数。其实这些得到的任务列表是有一定的排序规律的：最近得到运行的任务，将排序在 getRunningTasks()方法所返回的列表的表头位置。也就是说，越靠近列表的表头，则这个任务在开始运行时的时间距离现在的时间就越近。例如下面的演示代码：

```
/**
* 获取系统的任务信息，需要用户权限：android.permission.GET TASKS
*
* ActivityManager.RunningTaskInfo {
*     public ComponentName baseActivity  // 任务作为第一个活动的组件信息
*     public CharSequence description    // 任务当前状态的描述
*     public int id                      // 任务的 ID
*     public int numActivities           // 任务中所包含的活动的数目
*     public int numRunning              // 任务中处于运行状态的活动数目
*     public Bitmap thumbnail            // 任务当前状态的位图表示，目前为 null
*     public ComponentName topActivity       // 处于任务栈的栈顶的活动组件
* }
*/
package crazypebble.sysassist.taskmgr;

import crazypebble.sysassist.R;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;

import android.app.ActivityManager;
import android.app.ListActivity;
import android.app.ActivityManager.RunningTaskInfo;
import android.os.Bundle;
import android.widget.SimpleAdapter;

public class TaskMgrActivity extends ListActivity{

   private static List<RunningTaskInfo> taskList = null;
   private static final int maxTaskNum = 100;

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.task list);

      taskList = new ArrayList<RunningTaskInfo>();
      getTaskInfo();
```

```
            showTaskInfo();
        }

        public void showTaskInfo() {

            // 更新进程列表
            List<HashMap<String,String>> infoList = new
ArrayList<HashMap<String,String>>();
            for (Iterator<RunningTaskInfo> iterator = taskList.iterator();
iterator.hasNext();) {
                RunningTaskInfo taskInfo = iterator.next();
                HashMap<String, String> map = new HashMap<String, String>();
                map.put("task name", taskInfo.baseActivity.toString());
                map.put("task id", taskInfo.topActivity.toString());
                infoList.add(map);
            }

            SimpleAdapter simpleAdapter = new SimpleAdapter(
                    this,
                    infoList,
                    R.layout.task list item,
                    new String[]{"task name", "task id"},
                    new int[]{R.id.task name, R.id.task id} );
            setListAdapter(simpleAdapter);
        }

        public int getTaskInfo() {
            ActivityManager activityManager = (ActivityManager)
getSystemService(ACTIVITY SERVICE);
            taskList = activityManager.getRunningTasks(maxTaskNum);
            return taskList.size();
        }
    }
```

(3) 开始获取系统中的所有服务的信息。具体实现方法同上，需要调用 ActivityManager.getRunningServices(maxServiceNum)，参数 maxServiceNum 的含义与获取任务信息的含义是一样的，只不过在此不需要为用户添加任何权限而已。例如下面的演示代码：

```
    /**
    * ActivityManager.RunningServiceInfo {
    *    public long activeSince      // 服务第一次被激活的时间（启动和绑定方式）
    *    public int clientCount       // 连接到该服务的客户端数目
    *    public int clientLabel      // 【系统服务】为客户端程序提供用于访问标签
    *    public String clientPackage   // 【系统服务】绑定到该服务的包名
    *    public int crashCount         // 服务运行期间，出现 crash 的次数
    *    public int flags              // 服务运行的状态标志
    *    public boolean foreground     // 服务是否被作为前台进程执行
    *    public long lastActivityTime     // 该服务的最后一个活动的时间
    *    public int pid              // 非 0 值，表示服务所在的进程 Id
```




```
    *    public String process       // 服务所在的进程名称
    *    public long restarting      // 如果非 0，表示服务没有执行，将在参数给
定的时间点重启服务
    *    public ComponentName service    // 服务组件信息
    *    public boolean started      // 标识服务是否被显示的启动
    *    public int uid              // 拥有该服务的用户 Id
    * }
    */
   package crazypebble.sysassist.servicemgr;

   import crazypebble.sysassist.R;

   import java.util.ArrayList;
   import java.util.HashMap;
   import java.util.Iterator;
   import java.util.List;

   import android.app.ActivityManager;
   import android.app.ListActivity;
   import android.app.ActivityManager.RunningServiceInfo;
   import android.os.Bundle;
   import android.widget.SimpleAdapter;

   public class ServiceMgrActivity extends ListActivity{
      private static List<RunningServiceInfo> serviceList = null;
      private static final int maxServiceNum = 100;

      @Override
      public void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.service list);

         serviceList = new ArrayList<RunningServiceInfo>();
         getServiceInfo();

         showServiceInfo();
      }

      public void showServiceInfo() {

         // 更新进程列表
         List<HashMap<String,String>> infoList = new
ArrayList<HashMap<String,String>>();
         for (Iterator<RunningServiceInfo> iterator =
serviceList.iterator(); iterator.hasNext();) {
            RunningServiceInfo serviceInfo = iterator.next();
            HashMap<String, String> map = new HashMap<String, String>();
            map.put("service name", serviceInfo.service.toString());
            map.put("service id", serviceInfo.clientCount+"");
            infoList.add(map);
         }
```



```
            SimpleAdapter simpleAdapter = new SimpleAdapter(
                    this,
                    infoList,
                    R.layout.service list item,
                    new String[]{"service name", "service id"},
                    new int[]{R.id.service name, R.id.service id} );
            setListAdapter(simpleAdapter);
        }
        public int getServiceInfo() {
            ActivityManager activityManager = (ActivityManager)
getSystemService(ACTIVITY SERVICE);
            serviceList =
activityManager.getRunningServices(maxServiceNum);
            return serviceList.size();
        }
    }
```

这样就实现了我们需要的枚举功能，当然本程序并不能像其他安全管理软件那样，把应用程序的名字，图标等信息显示出来，而只是打印出来了一些包名信息。上述演示代码的目的是想读者讲解获取方法，读者可以以上述代码为基础进行扩展，实现自己需要的功能。

9.2.4　研究 Android 进程管理器的实现

在 9.2.3 节中介绍了枚举进程的方法，讲解了实现预期效果的原理和过程。在接下来的内容中，将继续以前面的预期效果为基础，介绍实现一个进程管理器的基本过程。

(1) 首先实现界面布局，在此采用选项卡式的布局，main.xml 的演示代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@android:id/tabhost"
android:layout width="fill parent"
android:layout height="fill parent">
<LinearLayout android:orientation="vertical"
android:layout width="fill parent"
android:layout height="fill parent"
android:padding="2dp">
<TabWidget android:id="@android:id/tabs"
android:layout width="fill parent"
android:layout height="wrap content" />
<FrameLayout android:id="@android:id/tabcontent"
android:layout width="fill parent"
android:layout height="fill parent"
android:padding="5dp" />
</LinearLayout>
</TabHost>
```

在上述演示代码中，每一个选项卡的内容都是一个列表 ListView，分别用于显示系统的进程、任务和服务列表，布局文件我们就此略过了。

(2) 在进程的详情中，使用不同背景色的 TextView 作为一个数据部分的标题，这样给

人视觉上一个比较清晰的层次感。进程详情文件 proc_detail.xml 的演示代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_height="wrap_content"
android:layout width="match parent"
android:scrollbars="none">
<LinearLayout
android:id="@+id/linearlayout1″
android:layout width="fill parent"
android:layout height="fill parent"
android:orientation="vertical"
android:paddingLeft="3dip"
android:paddingRight="3dip"
android:paddingTop="1dip"
android:paddingBottom="1dip">
<RelativeLayout android:layout width="match parent"
android:layout height="27dip"
android:background="#555555″
android:id="@+id/relativeLayout1″ >
<TextView android:textSize="7pt"
android:layout alignParentLeft="true"
android:id="@+id/textView1″
android:layout width="wrap content"
android:layout height="fill parent"
android:gravity="center vertical"
android:text="@string/detail process name"></TextView>
<Button android:id="@+id/btn_kill_process"
android:textSize="6pt"
android:layout alignParentRight="true"
android:layout width="wrap content"
android:layout height="fill parent"
android:paddingTop="0dip"
android:paddingBottom="0dip"
android:paddingRight="10dip"
android:paddingLeft="10dip"
android:layout marginTop="2dip"
android:textStyle="bold"
android:text="@string/kill_process"
android:textColor="#5555EE"></Button>
</RelativeLayout>
<TextView
android:id="@+id/detail process name"
android:layout height="wrap content"
android:layout width="fill parent"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:text="@string/detail process copyright"
android:textSize="7pt"
```



```
android:layout alignParentLeft="true"
android:layout width="fill parent"
android:layout height="25dip"
android:gravity="center vertical"
android:background="#555555"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:id="@+id/detail process copyright"
android:layout height="wrap content"
android:layout width="fill parent"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:text="@string/detail process install dir"
android:textSize="7pt"
android:layout alignParentLeft="true"
android:layout width="fill parent"
android:layout height="25dip"
android:gravity="center vertical"
android:background="#555555"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:id="@+id/detail process install dir"
android:layout height="wrap content"
android:layout width="fill parent"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:text="@string/detail process data dir"
android:textSize="7pt"
android:layout alignParentLeft="true"
android:layout width="fill parent"
android:layout height="25dip"
android:gravity="center vertical"
android:background="#555555"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:id="@+id/detail process data dir"
android:layout height="wrap content"
android:layout width="fill parent"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:text="@string/detail process package size"
android:textSize="7pt"
android:layout alignParentLeft="true"
android:layout width="fill parent"
android:layout_height="25dip"
```

```
android:gravity="center vertical"
android:background="#555555″
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:id="@+id/detail process package size"
android:layout height="wrap content"
android:layout width="fill parent"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:text="@string/detail process permission"
android:textSize="7pt"
android:layout alignParentLeft="true"
android:layout width="fill parent"
android:layout height="25dip"
android:gravity="center vertical"
android:background="#555555″
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:id="@+id/detail process permission"
android:layout height="wrap content"
android:layout width="fill parent"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:text="@string/detail process service"
android:textSize="7pt"
android:layout alignParentLeft="true"
android:layout width="fill parent"
android:layout height="25dip"
android:gravity="center vertical"
android:background="#555555″
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:id="@+id/detail process service"
android:layout height="wrap content"
android:layout width="fill parent"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
<TextView
android:text="@string/detail process activity"
android:textSize="7pt"
android:layout alignParentLeft="true"
android:layout width="fill parent"
android:layout height="25dip"
android:gravity="center vertical"
android:background="#555555″
android:paddingTop="3dip"
```

```
android:paddingBottom="3dip" />
<TextView
android:id="@+id/detail process activity"
android:layout height="wrap content"
android:layout width="fill parent"
android:paddingTop="3dip"
android:paddingBottom="3dip" />
</LinearLayout>
</ScrollView>
```

上述演示代码中的整个详情信息是一个 ScrollView，在第一行中嵌入了一个 Button，其他行的数据显示都比较简单。

(3) 开始获取进程的图标、进程名、Application 名字和 CPU 内存信息。在演示文件 BasicProgramUtil.java 中，使用了类 BasicProgramUtil 来存放进程列表中显示的摘要信息，包括了进程图标、进程名、Application 名、CPU 和内存信息。

```
/**
* 应用程序包的简要信息
*/
package crazypebble.sysassist.procmgr;
import android.graphics.drawable.Drawable;
public class BasicProgramUtil{
/*
* 定义应用程序的简要信息部分
*/
private Drawable icon;                              // 程序图标
private String programName;                          // 程序名称
private String processName;
private String cpuMemString;
public BasicProgramUtil() {
icon = null;
programName = "";
processName = "";
cpuMemString = "";
}
public Drawable getIcon() {
return icon;
}
public void setIcon(Drawable icon) {
this.icon = icon;
}
public String getProgramName() {
return programName;
}
public void setProgramName(String programName) {
this.programName = programName;
}
public String getCpuMemString() {
return cpuMemString;
}
```

```
public void setCpuMemString(String cpuMemString) {
this.cpuMemString = cpuMemString;
}
```

9.3 将 Android 软件从手机内存转移到存储卡

Android 系统只能把软件安装在手机内存里，使本来就不大的手机内存显得捉襟见肘。其实手机中的存储器分为两种：随机存储器(RAM)和只读存储器(ROM)。其中手机 ROM 相当于 PC 上的硬盘，用于存储手机操作系统和软件，也叫 FLASH ROM，决定手机存储空间的大小。手机 RAM 相当于 PC 的内存，其大小决定手机的运行速度。我们完全可以将 Android 软件从手机内存转移到存储卡，这样可以达到系统优化的目的。要想把 Android 系统中的软件安装到 SD 卡上，只需要本节介绍的三步工作即可。

9.3.1 第一步：准备工作

在进行转移工作之前，需要先判断这个程序是否可以转移。例如在下面的演示代码中，通过 PackageManager 得到该程序的权限列表，然后通过权限的名字得到该权限的 PermissionInfo。

```
private void getPermisson(Context context) {
    try {
        PackageManager pm = context.getPackageManager();
        PackageInfo pi = pm.getPackageInfo(context.getPackageName(), 0);
        // 得到自己的包名
        String pkgName = pi.packageName;
        PackageInfo pkgInfo = pm.getPackageInfo(pkgName,
                PackageManager.GET_PERMISSIONS);//通过包名，返回包信息
        String sharedPkgList[] = pkgInfo.requestedPermissions;//得到权限列表
        for (int i = 0; i < sharedPkgList.length; i++) {
            String permName = sharedPkgList[i];
            PermissionInfo tmpPermInfo = pm.getPermissionInfo(permName,
0);//通过 permName 得到该权限的详细信息
            PermissionGroupInfo pgi = pm.getPermissionGroupInfo(
                    tmpPermInfo.group, 0);//权限分为不同的群组，通过权限名，我
们得到该权限属于什么类型的权限。
            tv.append(i + "-" + permName + "\n");
            tv.append(i + "-" + pgi.loadLabel(pm).toString() + "\n");
            tv.append(i + "-" + tmpPermInfo.loadLabel(pm).toString()+ "\n");
            tv.append(i + "-" +
tmpPermInfo.loadDescription(pm).toString()+ "\n");
            tv.append(mDivider + "\n");
        }
    } catch (NameNotFoundException e) {
        Log.e("##ddd", "Could'nt retrieve permissions for package");
    }
}
```

通过上述代码，就成功判断了是否可以将某个软件移动到 SD 卡上。但是还是有如下两个问题。

(1) 能移动到 SD 卡上面的程序都是 Android 2.2 以后 API 才支持这个功能，检测该程序的开发版本是否是 2.2 后或者 2.2 以上开发版本就可以判断是否提供可移动到 SD 卡功能。

(2) 怎么移动？依次进入【Android 系统设置】|【应用程序】|【管理应用程序】列表下，列出了系统已安装的应用程序。选择其中一个程序，则进入【应用程序信息(Application Info)】界面。这个界面显示了程序名称、版本、存储、权限等信息，并有卸载、停止、清除缓存等按钮，可谓功能不少。如果在编写相关程序时(比如任务管理器)可以调用这个面板，自然提供了很大的方便。那么如何实现呢？

在 Android SDK 2.3(API Level 9)以上版本中，提供了如下文档路径的接口：

```
docs/reference/android/provider
```

由此可见，只要以 android.provider.Settings.ACTION_APPLICATION_DETAILS_SETTINGS 作为 Action；“package 应用程序的包名”作为 URI，就可以用 startActivity 启动应用程序信息界面了。代码如下：

```
Intent intent = new Intent(Settings.ACTION APPLICATION DETAILS SETTINGS);
Uri uri = Uri.fromParts(SCHEME, packageName, null);
intent.setData(uri);
startActivity(intent);
```

但是，在 Android 2.3 之前的版本，并没有公开相关的接口。通过查看系统设置 platform/packages/apps/Settings.git 程序的源代码，可以发现应用程序信息界面为 InstalledAppDetails。

接下来分别以 Android 2.1 和 Android 2.2 为例，研究它们的应用管理程序(ManageApplications.java)是如何启动 InstalledAppDetails 的。

```
// utility method used to start sub activity
private void startApplicationDetailsActivity() {
// Create intent to start new activity
Intent intent = new Intent(Intent.ACTION VIEW);
intent.setClass(this, InstalledAppDetails.class);
intent.putExtra(APP PKG NAME, mCurrentPkgName);
// start new activity to display extended information
startActivityForResult(intent, INSTALLED APP DETAILS);
}
```

但是常量 APP_PKG_NAME 的定义并不相同。

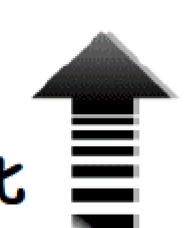
在 Android 2.2 中定义为 pkg，在 Android 2.1 中定义为 com.android.settings.ApplicationPkgName。那么对于在 Android 2.1 及以下版本，我们可以这样调用 InstalledAppDetails：

```
Intent i = new Intent(Intent.ACTION VIEW);
i.setClassName("com.android.settings","com.android.settings.InstalledApp
Details");
```

```
i.putExtra("com.android.settings.ApplicationPkgName", packageName);
startActivity(i);
```

对于在 Android 2.2，只需替换上面 putExtra 的第一个参数为"pkg"。由此可见，通用的调用“应用程序信息”的代码如下：

```
    private static final String SCHEME = "package";
    /**
     * 调用系统 InstalledAppDetails 界面所需的 Extra 名称(用于 Android 2.1 及之前
版本)
     */
    private static final String APP_PKG_NAME_21 =
"com.android.settings.ApplicationPkgName";
    /**
     * 调用系统 InstalledAppDetails 界面所需的 Extra 名称(用于 Android 2.2)
     */
    private static final String APP_PKG_NAME_22 = "pkg";
    /**
     * InstalledAppDetails 所在包名
     */
    private static final String APP_DETAILS_PACKAGE_NAME =
"com.android.settings";
    /**
     * InstalledAppDetails 类名
     */
    private static final String APP_DETAILS_CLASS_NAME =
"com.android.settings.InstalledAppDetails";
    /**
     * 调用系统 InstalledAppDetails 界面显示已安装应用程序的详细信息。对于 Android 2.3
(Api Level 9)以上，使用 SDK 提供的接口； 2.3 以下，使用非公开的接口(查看
InstalledAppDetails 源码)。
     *
     * @param context
     *
     * @param packageName
     *            应用程序的包名
     */
    public static void showInstalledAppDetails(Context context, String
packageName) {
        Intent intent = new Intent();
        final int apiLevel = Build.VERSION.SDK_INT;
        if (apiLevel >= 9) { // 2.3(ApiLevel 9)以上，使用 SDK 提供的接口
            intent.setAction(Settings.ACTION_APPLICATION_DETAILS_SETTINGS);
            Uri uri = Uri.fromParts(SCHEME, packageName, null);
            intent.setData(uri);
        } else { // 2.3 以下，使用非公开的接口(查看 InstalledAppDetails 源码)
            //在 2.2 和 2.1 中，InstalledAppDetails 使用的 APP_PKG_NAME 不同。
            final String appPkgName = (apiLevel == 8 ? APP_PKG_NAME_22
                    : APP_PKG_NAME_21);
            intent.setAction(Intent.ACTION_VIEW);
            intent.setClassName(APP_DETAILS_PACKAGE_NAME,
```



```
                APP DETAILS CLASS NAME);
            intent.putExtra(appPkgName, packageName);
        }
        context.startActivity(intent);
    }
```

9.3.2　第二步：存储卡分区

首先我们需要对手机 SD 卡进行分区，分一个 FAT32 分区和一个 Ext3 分区，FAT32 分区用于正常存储图片、音乐、视频等资料，而 Linux 格式的 Ext3 分区就是用于扩容安装软件的分区。以笔者的 2G SD 卡为例，FAT32 分区 1.35GB，Ext3 分区 494MB。下载并安装 Acronis Disk Director Suite 软件。将手机 SD 卡装入读卡器并连接电脑，然后运行 Acronis Disk Director Suite 软件。

(1) FAT32 分区

找到代表 SD 卡的磁盘分区，单击右键，选择【删除】命令，删除已有分区。当成为“未分配”分区时，单击右键，选择【创建分区】，在弹出的对话框中，【文件系统】选择 FAT32，创建为“主分区”，设置分区大小 1.35GB，单击【确定】按钮。

(2) Ext3 分区

在剩余的 494MB 分区上，单击右键，选择【创建分区】命令，在弹出的对话框中，【文件系统】选择 Ext3，创建为“主分区”，设置分区大小 494MB，单击【确定】按钮。

(3) 确认分区

上述分区设定完成后，软件只是记录了分区操作，并没有真正在 SD 卡上进行分区。单击软件工具栏中的【提交】按钮，确认执行分区操作，提示“操作成功完成”，说明分区成功了。

9.3.3　第三步：将软件移动到 SD 卡

在存储卡分区工作完成后，只需要把系统默认的软件安装目录/data/app 转移到 SD 卡的 Ext3 分区上，然后通过 ln 命令建立软链接，使系统自动把软件安装到 SD 卡上，达到节省手机内存空间的目的。

(1) 将存储卡装回手机，重新启动，使系统识别到 Ext3 分区。在手机上运行超级终端，依次输入以下命令来验证系统是否识别了 Ext3 分区：

- su：提示高级权限授权，选择【总是同意】。
- busybox df －h：如果显示的列表中有/dev/block/mmcblk0p2 的信息说明系统已成功识别了 Ext3 分区，如图 9-9 所示。

(2) 依次输入以下命令将/data/app 目录转移到 SD 卡的 Ext3 分区上。

- cp －a /data/app /system/sd/：将/data/app 目录复制到/system/sd/下。
- rm －r /data/app：删除/data/app 目录。
- ln －s /system/sd/app /data/app：建立软链接。
- Reboot：重启手机。

如图 9-10 所示。

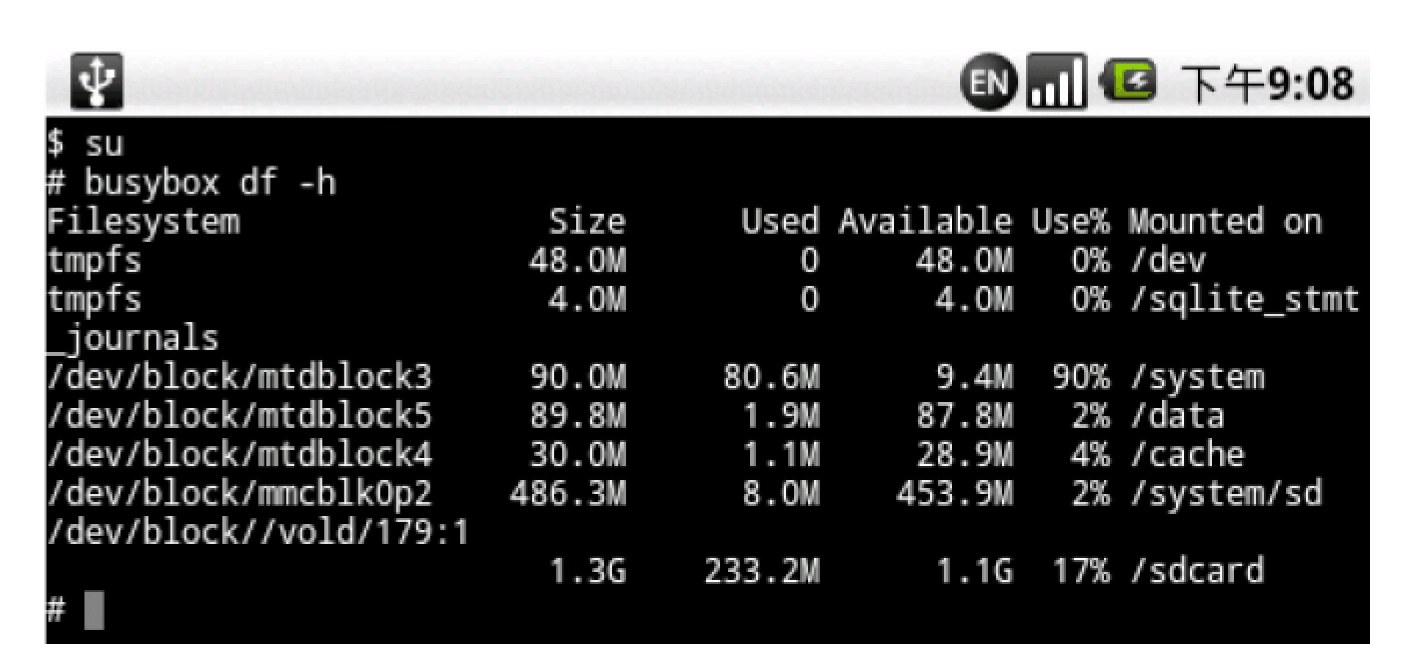


图 9-9　识别了 Ext3 分区

```
$ su
# cp -a /data/app /system/sd/
# rm -r /data/app
# ln -s /system/sd/app /data/app
# reboot
```

图 9-10　分区界面

(3) 此时重启之后，手机上安装的所有软件就全部转移到了 SD 卡上，看看你的手机可用空间是不是增大了。以后再安装软件也是直接安装到 SD 卡上，不用担心空间不足的问题了。而且这样做还有一个好处，刷新 ROM 后，以前安装过的软件并没有被清除，还保存在 SD 卡上，输入下列命令就可以轻松恢复，就不用再一一安装了，非常方便、实用。

- su：取得高级权限。
- cd /data：进入/data 目录。
- cp －a app /system/sd/app：将 app 目录中的内容复制到/system/sd/app 目录。
- rm －r app：删除 app 目录。
- ln －s /system/sd/app /data/app：建立软链接。
- reboot：重新启动。

如图 9-11 所示。

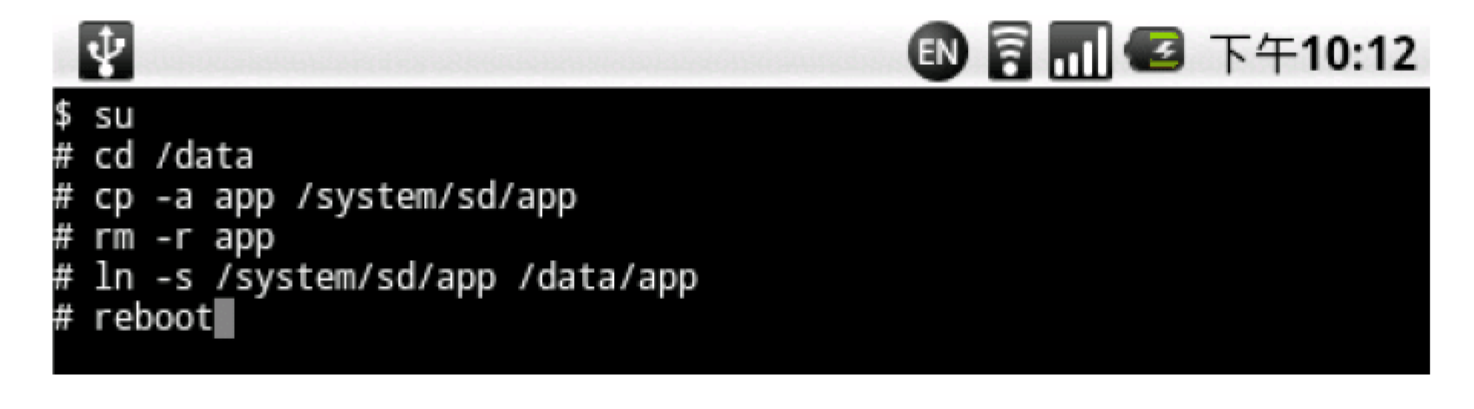


图 9-11　操作界面

扩容之后，笔者用真机进行了测试。发现如果刷新 ROM 后未安装任何软件，手机可用空间为 87MB。当安装若干软件后，可用空间下降为 73MB。将软件目录转移到 SD 卡上后，可用空间变为 80MB。可能有的读者会有疑惑，为什么没恢复到 87MB 呢？这是因为我们只是将软件移动到了 SD 卡上，而软件的缓存数据仍然会占用手机内存，所以手机内存还是会下降。当然软件的缓存数据也可以移动到 SD 卡上，但这样会拖慢软件运行速

度，所以不推荐大家使用。

注意： 当将软件移动到 SD 卡上后，原有的部分桌面插件会无法正常显示，删除后，重新加入桌面即可。另外，SD 卡的 Ext3 分区可以视为手机硬件的一部分，移除 SD 卡后，安装的软件将无法运行。插入 SD 卡，重新启动手机即可正常使用。

9.4　常用的系统优化工具

在当前市面中，已经诞生了很多系统优化工具。我们使用这些第三方优化工具，可以优化我们 Android 设备。在本节的内容中，将简要介绍市面中两款常用的优化工具。

9.4.1　优化大师

优化大师是一款功能强大的手机系统优化软件，它提供了全面有效且简便安全的手机体检、开机加速、批量卸载、文件管理四大功能模块及数个附加的工具软件。使用 Gphone 优化大师，能够有效地帮助用户了解自己的手机软硬件信息；提升手机开机速度；扫描有危险的软件；维护手机的正常运转。Android 优化大师的运行界面如图 9-12 所示。



图 9-12　Android 优化大师

Android 优化大师的主要功能如下所示：

- 手机体检：为我们的手机进行全面的安全体检扫描，让一切不安全因素无所遁形。
- 使用统计：为我们提供手机的使用记录日志，让您完全掌握手机的使用情况。
- 电池统计：手机电池、CPU、网络使用率、GPS 使用率、传感器、Wi-Fi 等手机状态的全面分析。
- 软件分析：快速对手机中的软件进行分析，让您更好地管理手机中的软件。

- 开机加速：扫描手机中开机自动启动程序，优化您手机的开机速度。
- 程序管理：方便快捷地管理手机中的已安装程序和系统程序。
- 进程管理：管理系统运行的进程，使您的手机保持在最佳的使用状态。
- 批量卸载：快速批量卸载手机中您想要卸载的软件，告别麻烦，节省时间。
- 文件浏览：有了优化大师文件浏览，无须再安装一款文件浏览器软件。
- 软件检测：检测手机中是否装有恶意软件，并给出解决方案。
- 手机信息：最全面的手机信息，手机的各项信息全部为您呈现。

9.4.2　360 优化大师

360 优化大师是一款专业的安卓手机优化工具，全方位优化管理手机，提高运行速度，改善手机使用效率，功能全面强大，是您手机系统飞速运转的助推剂。界面效果如图 9-13 所示。



图 9-13　360 优化大师

Android 版 360 优化大师的主要功能如下。

(1) 手机加速：一个好汉三个帮

360 优化大师界面简洁，“手机加速”被放在了最显眼的位置。单击后，360 优化大师会自动开始扫描。扫描结果显示，需要深度清理的垃圾竟然有将近 100M，主要来自于新浪微博和 UC 下载记录。

俗话说一个好汉三个帮，“手机加速”还辅有“进程管理”、“缓存清理”、“深度清理”三个帮手，久而久之累积的手机垃圾交给“深度清理”和“缓存清理”，而那些烦人的随机启动、后台关不掉的进程就交给“进程管理”搞定吧——依照自己的需求，只保留必要的常用软件，可以禁止所有不需要的。

(2) 节电优化：不惧安卓电老虎

安卓的电池续航能力一直被人诟病，节电是安卓用户的一大需求。360 优化大师“节



电优化”界面上方显示了电池的剩余电量和健康状况，下方是“背景数据”和“自动同步”两大耗电大户，可以方便地选择关闭。更多设置中，陈列了所有可能谋杀电池的设备，比如 Wi-Fi、蓝牙、GPS、屏幕亮度等，可以参考 360 的节电建议进行设置。

(3) 文件管理：方便分类管理海量文件

Android 手机本身没有文件管理工具，所以新用户经常不知道如何传文件，也不知道下载的安装程序去了哪里。360 优化大师里自带的“文件管理”功能，可以按图片、音乐、视频、文档、压缩包、安装包六大类别查看文件，并进行传输、删除、移动等操作，还支持蓝牙和邮件等途径与朋友分享。同时，还会显示储存卡当前的使用空间。

(4) 快捷设置：化繁为简

当新用户拿到一步 Android，怎么调铃声？怎么设置上网？怎么与电脑互联？这都是急需解决的问题。360 优化大师看到了这一需求，“快捷设置”功能将各种手机设置集合、分类，并简化了部分复杂功能，可以一键设置上网，让 Android 新手轻松上手，十分贴心。

(5) 系统检测：检验 Android 硬件配置

除了软件设置之外，360 优化大师“系统检测”还可以检验 Android 手机的硬件配置，支持查看系统版本、手机串号、CPU、内存、屏幕分辨率等硬件配置信息，而且还能够监测 SD 卡速，测试屏幕暗点亮点等问题，对于新手买手机时，倒是很不错的辅助工具，避免买到以次充好的手机而上当受骗。

第 10 章

开发一个 Android 优化系统

前面曾经讲解过开发 Android 进程管理器的原理和演示代码，本章将通过一个综合实例的实现过程来讲解开发 Android 优化系统的基本流程。本章源码保存在网络资源 daima\10\文件夹中。

10.1 优化大师介绍

手机优化大师的功能是，通过电脑端和手机端分别实现对 Android 手机操作系统的管理和性能优化。目前，PC 版本可以在电脑上管理手机中的通讯录、短信、应用程序和音乐等，同时通过任务管理、系统清理等功能可实现对手机性能的优化。手机端版本使用了插件式的设计，可以通过设置选项中的插件安装，根据用户的喜好选择不同的功能，安装向导就会让大家选择自己所需的功能。市面中常见的 Android 优化系统是本书 9.4.1 节中讲解的优化大师，其中分为手机端和 PC 端。

10.1.1 手机优化大师客户端

Android 手机优化大师的客户端是一款运行在 Android 手机上的系统增强优化软件，目的是为用户提供便利、绿色且免费的服务，具备软件管理、任务管理、系统清理、文件管理、条码扫描等十余种功能，覆盖了日常使用的方方面面。使用手机优化大师，能够有效提高系统的整体使用效率和稳定性，帮助用户全方位管理好自己的手机。客户端的界面效果如图 10-1 所示。



图 10-1 Android 手机优化大师的客户端

10.1.2 手机优化大师 PC 端

手机优化大师 PC 版是一款强大的智能手机管理工具，其中 1.0 版本全面兼容基于 Android 内核的 Android 手机的管理，可以完美运行在 Windows XP/Vista 和 Windows 7 操作系统，它提供了如下功能。

- 软件商店：应用软件、游戏下载。





- 联系人管理：在电脑上添加、删除、修改联系人及相关归属地显示。
- 短信管理：在电脑上查看、发送、删除短信。
- 通话记录：批量删除、查看通话记录。
- 闹铃管理：对部分固件的 Android 手机闹铃提供了新增、删除、修改提醒的支持。
- 文件管理：在电脑上浏览手机 SD 卡或文件系统的内容。
- 书签管理：管理手机系统自带浏览器的收藏夹内容。
- 应用管理：在电脑上查看手机已装软件或游戏，提供批量删除版本检测等操作。
- 系统信息：提供较为全面的手机硬件、软件系统信息查看。
- 手机设置：在电脑上开关 Android 设备的 Wi-Fi、蓝牙、重力感应等。
- 系统清理：自动扫描 Android 系统的运行临时文件和缓存。
- 启动管理：提供数百项软件自启动检测，轻松查找恶意软件。
- 屏幕截图：在电脑上截取手机屏幕，支持保存为 GIF、JPG、PNG 和 BMP 格式。
- 任务管理：查看手机上当前运行的应用和内存占用情况。
- APK 安装：内置了强大的 APK 安装器，可以检测 APK 文件中是否包含广告和安全威胁。

PC 端版本的界面效果如图 10-2 所示。



图 10-2　PC 端版本的界面效果

10.2　项 目 介 绍

本项目实例的功能是开发一个简易版的 Android 优化系统，可以实现进程维护管理和文件管理。为了使整个项目与上一节中介绍的优化大师类似，还设置了其他功能，例如：

手机体验、程序管理、网络管理、安装卸载、垃圾清理、节电管理和优化设置。读者可以在本实例的基础上进行扩充，实现上面的其他功能。

本项目的实现流程如图 10-3 所示。

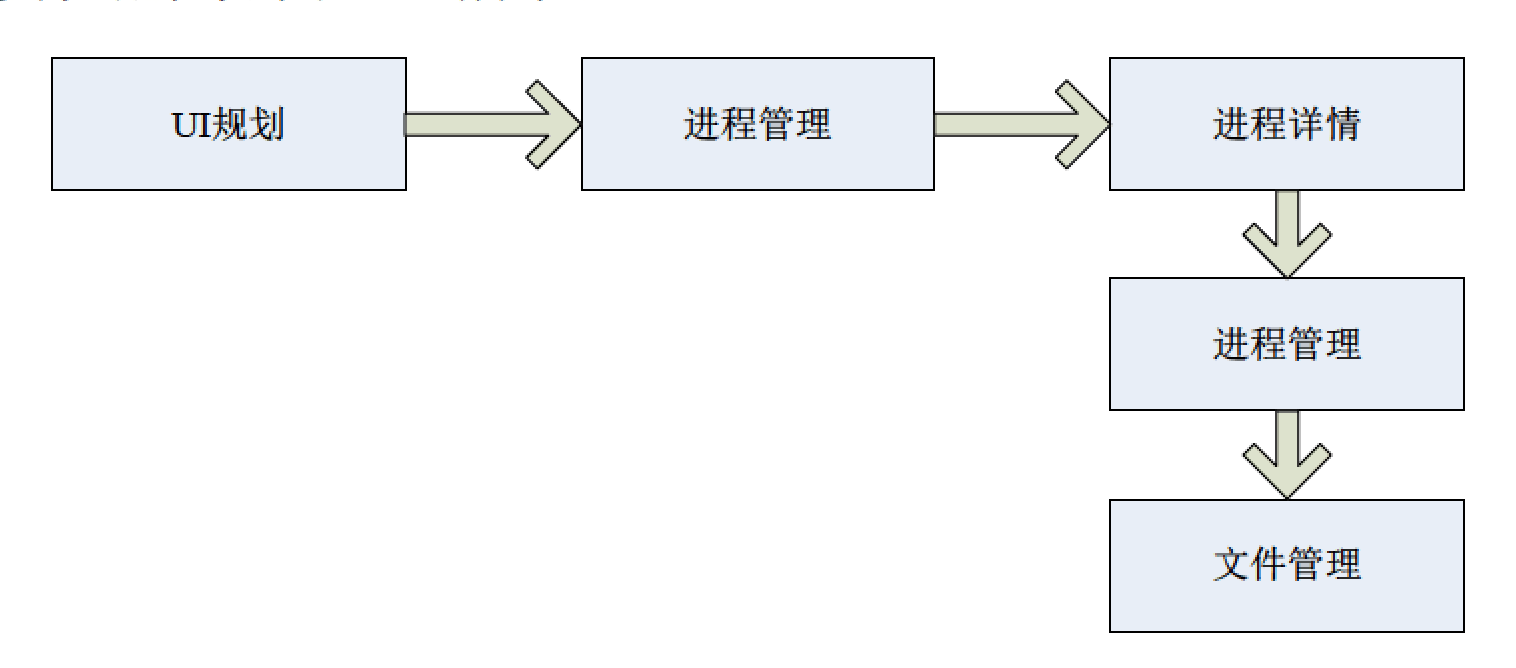


图 10-3　实现流程

10.2.1　规划 UI 界面

为了后期的升级考虑，本项目一共有 9 个模块，所以 UI 界面也包括 9 个模块主界面，UI 结构如图 10-4 所示。

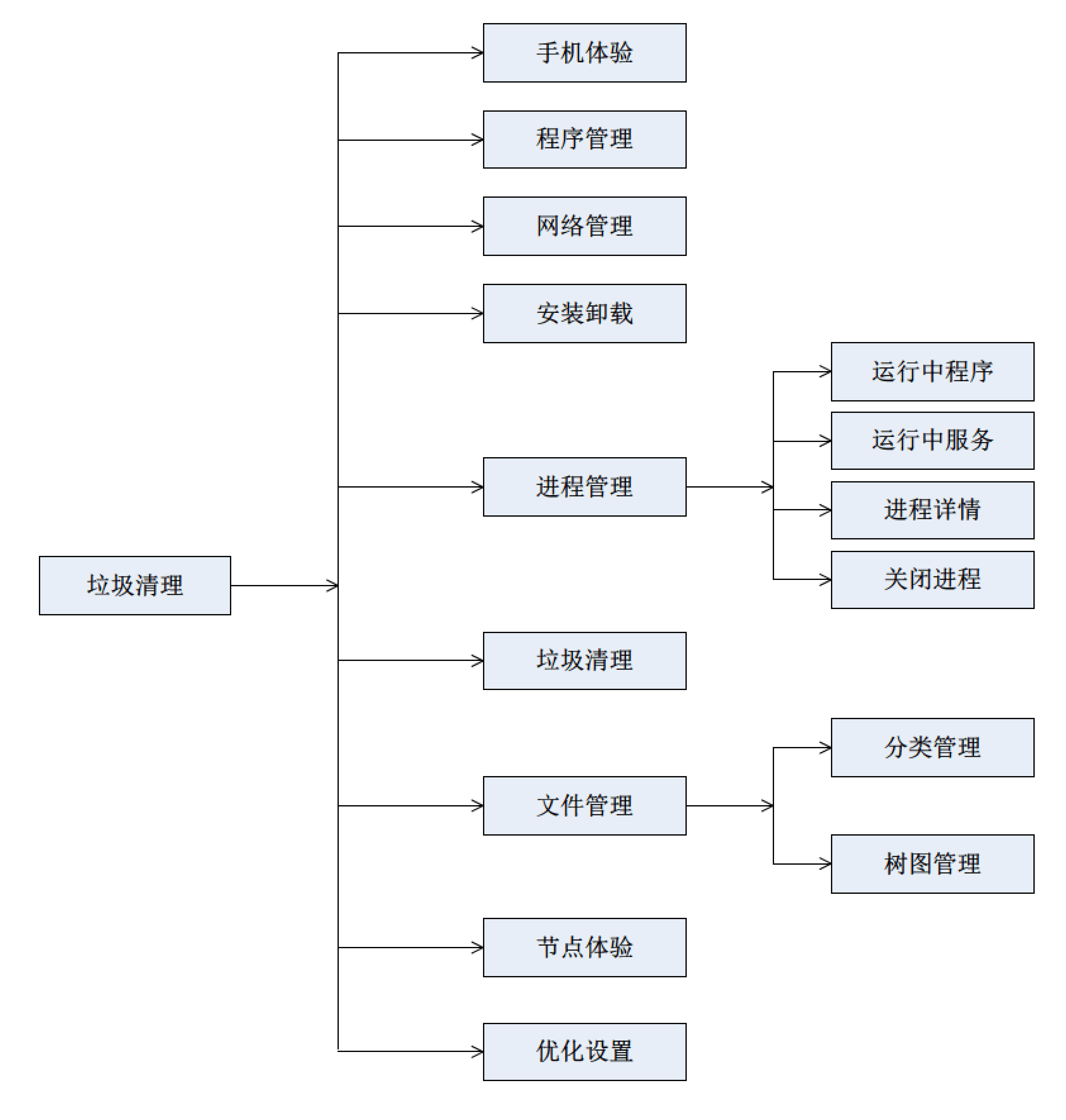


图 10-4　UI 界面结构

10.2.2　预期效果

本实例的主界面效果如图 10-5 所示。

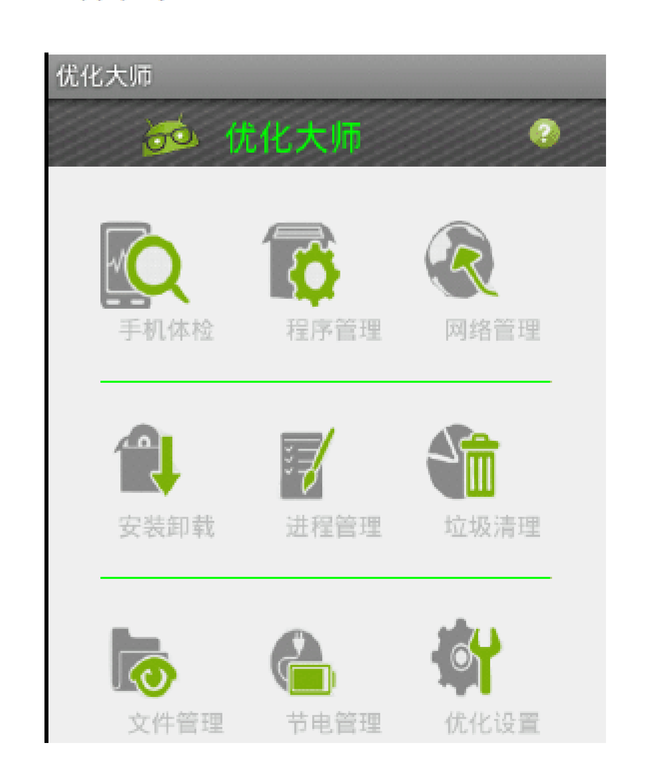


图 10-5　主界面执行效果

10.3　准 备 工 作

到此为止，一个项目的准备工作就做好了。接下来将介绍本项目的具体实现过程，希望读者认真体会每一段代码的功能和编写原理，为提高自己的开发水平做好准备。

10.3.1　新建工程

打开 Eclipse，依次选择 File | New | Android Project 命令，新建一个名为“AndroidManager”的工程文件，如图 10-6 所示。

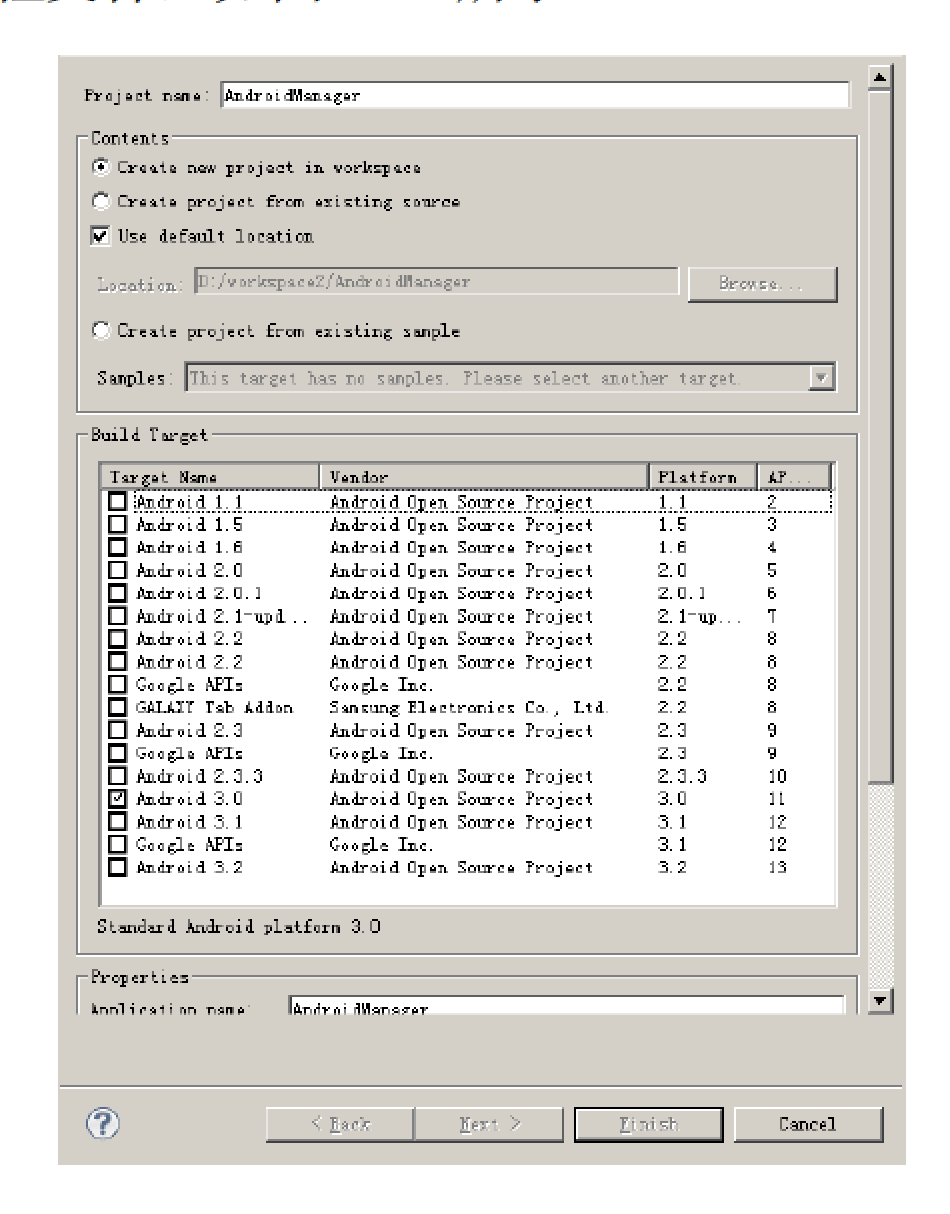


图 10-6　新建工程

10.3.2 主界面

主界面即项目执行后首先显示的界面，实现本项目主界面的流程如下。

(1) 编写主布局文件 main.xml，主要代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabhost" android:layout width="fill parent"
    android:layout height="fill parent">
    <LinearLayout android:layout width="fill parent"
android:orientation="vertical"
        android:layout height="fill parent" android:padding="5dp">
        <FrameLayout android:id="@android:id/tabcontent"
                    android:padding="5dp"
                    android:layout width="fill parent"
                    android:layout weight="1.0"
                   android:layout height="0.0dip" />
        <TabWidget android:id="@android:id/tabs"
                  android:layout width="fill parent"
                  android:visibility="gone"
                 android:layout height="wrap content" />
        <RadioGroup android:orientation="horizontal"
android:id="@+id/main radio" android:gravity="bottom"
android:layout width="fill parent" android:layout height="wrap content"
android:layout weight="0.0">
            <RadioButton android:id="@+id/btn1"
android:layout marginTop="2.0dip" android:tag="btn1"
android:layout width="wrap content" android:layout height="wrap content"
android:drawableTop="@drawable/process" android:text="进程"
style="@style/main tab bottom"/>
            <RadioButton android:id="@+id/btn2"
android:layout marginTop="2.0dip" android:tag="btn2"
android:layout width="wrap content" android:layout height="wrap content"
android:drawableTop="@drawable/task" android:text="任务"
style="@style/main tab bottom"/>
            <RadioButton android:id="@+id/btn3" android:layout marginTop=
"2.0dip" android:tag="btn3" android:layout width="wrap content" android:layout
height="wrap content" android:drawableTop="@drawable/service" android:text=
"服务" style="@style/main tab bottom"/>
            <RadioButton android:id="@+id/btn4" android:layout marginTop=
"2.0dip" android:tag="btn4" android:layout width="wrap content" android:layout
height="wrap content" android:drawableTop="@drawable/chart" android:text=
"图表" style="@style/main_tab_bottom"/>
            <RadioButton android:id="@+id/btn5" android:layout marginTop=
"2.0dip" android:tag="btn5" android:layout width="wrap content" android:layout
height="wrap content" android:drawableTop="@drawable/filebtn" android:text=
"文件" style="@style/main tab bottom"/>
        </RadioGroup>
    </LinearLayout>
</TabHost>
```



上述布局文件比较简单，核心功能是 RadioGroup 控件，笔者进行的是最少层级优化处理。

(2) 编写布局文件 nine_grid.xml，此文件的功能是实现九宫效果功能，将整个界面分成 3 行 3 列的效果。具体代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="fill parent"
android:background="@drawable/list bg"
    android:layout height="fill parent">
    <RelativeLayout android:id="@+id/toplayout" android:background=
"@drawable/topbg"
        android:layout width="fill parent" android:layout height="wrap content">
        <ImageView android:id="@+id/topimg" android:layout width=
"wrap content" android:layout marginLeft="50dp"
android:layout marginTop="8dp"
            android:layout height="wrap content"
android:src="@drawable/glass"/>
        <TextView android:id="@+id/title" android:layout toRightOf=
"@id/topimg" android:layout width="wrap content"
android:gravity="center" android:layout marginLeft="10dp"
            android:text="优化大师" android:layout height="wrap content"
android:layout marginTop="8dp" android:textSize="20dp"
android:textAppearance="?android:attr/textAppearanceLarge"
android:textColor="#00ff00"/>
        <ImageView android:id="@+id/question"
android:layout alignParentRight="true"
android:layout width="wrap content" android:layout marginRight="20dp"
android:layout marginTop="8dp"
            android:layout height="wrap content"
android:src="@drawable/question"></ImageView>
    </RelativeLayout>
    <ImageView android:id="@+id/advertise"
android:layout width="wrap content"
        android:layout alignParentBottom="true"
        android:layout alignParentLeft="true"
android:layout height="wrap content"
        android:src="@drawable/bottomlogo" />

    <TableLayout android:clickable="true" android:focusable="true"
android:layout height="fill parent" android:layout width="wrap content"
android:layout marginTop="70dp" android:paddingLeft="30dp"
android:paddingRight="30dp">
        <TableRow android:layout height="fill parent"
android:layout width="wrap content">
            <LinearLayout android:id="@+id/checkhealth"
android:layout height="wrap content" android:layout width="wrap content"
android:orientation="vertical">
                <ImageView android:src="@drawable/checkhealth"
android:layout width="wrap content" android:layout height="wrap content"
android:layout_alignParentTop="true"/>
```




```
                <TextView android:text="手机体检"
android:layout width="wrap content" android:layout height="wrap content"
android:layout marginLeft="10dp" android:gravity="center horizontal"/>
            </LinearLayout>
            <LinearLayout android:id="@+id/proadmin"
android:layout height="wrap content" android:layout width="wrap content"
android:orientation="vertical" android:layout marginLeft="20dp">
                <ImageView android:src="@drawable/proadmin"
android:layout width="wrap content" android:layout height="wrap content"
android:layout alignParentTop="true"/>
                <TextView android:text="程序管理"
android:layout width="wrap content" android:layout height="wrap content"
android:layout marginLeft="15dp" android:gravity="center horizontal"/>
            </LinearLayout>
            <LinearLayout android:id="@+id/netadmin"
android:layout height="wrap content" android:layout width="wrap content"
android:orientation="vertical" android:layout marginLeft="20dp">
                <ImageView android:src="@drawable/netadmin"
android:layout width="wrap content" android:layout height="wrap content"
android:layout alignParentTop="true"/>
                <TextView android:text="网络管理"
android:layout width="wrap content" android:layout height="wrap content"
android:layout marginLeft="15dp" android:gravity="center horizontal"/>
            </LinearLayout>
        </TableRow>
        <View android:layout height="1dip"
android:layout width="fill parent" android:background="#00FF00"
android:layout marginTop="20dp" android:layout marginBottom="20dp"/>
        <TableRow android:layout height="fill parent"
            android:layout width="wrap content">
            <LinearLayout android:id="@+id/install"
android:layout height="wrap content" android:layout width="wrap content"
android:orientation="vertical">
                <ImageView android:src="@drawable/install"
android:layout width="wrap content" android:layout height="wrap content"
android:layout alignParentTop="true"/>
                <TextView android:text="安装卸载"
android:layout width="wrap content" android:layout height="wrap content"
android:layout marginLeft="10dp" android:gravity="center horizontal"/>
            </LinearLayout>
            <LinearLayout android:id="@+id/adminpro"
android:layout height="wrap content" android:layout width="wrap content"
android:orientation="vertical" android:layout marginLeft="20dp">
                <ImageView android:src="@drawable/adminpro"
android:layout width="wrap content" android:layout height="wrap content"
android:layout alignParentTop="true" />
                <TextView android:text="进程管理"
android:layout width="wrap content" android:layout height="wrap content"
android:layout marginLeft="15dp" android:gravity="center horizontal"/>
            </LinearLayout>
            <LinearLayout android:id="@+id/clear"
```

```
android:layout height="wrap content" android:layout width="wrap content"
android:orientation="vertical" android:layout marginLeft="20dp">
                <ImageView android:src="@drawable/clear"
android:layout width="wrap content" android:layout height="wrap content"
android:layout alignParentTop="true"/>
                <TextView android:text="垃圾清理"
android:layout width="wrap content" android:layout height="wrap content"
android:layout marginLeft="15dp" android:gravity="center horizontal"/>
            </LinearLayout>
        </TableRow>
        <View android:layout height="1dip"
android:layout width="fill parent" android:background="#00FF00"
android:layout marginTop="20dp" android:layout marginBottom="20dp"/>
        <TableRow android:layout height="fill parent"
            android:layout width="wrap content">
            <LinearLayout android:id="@+id/fileadmin"
android:layout height="wrap content" android:layout width="wrap content"
android:orientation="vertical">
                <ImageView android:src="@drawable/fileadmin"
android:layout width="wrap content" android:layout height="wrap content"
android:layout alignParentTop="true"/>
                <TextView android:text="文件管理"
android:layout width="wrap content" android:layout height="wrap content"
android:layout marginLeft="15dp" android:gravity="center horizontal"/>
            </LinearLayout>
            <LinearLayout android:id="@+id/batteryadmin"
android:layout height="wrap content" android:layout width="wrap content"
android:orientation="vertical" android:layout marginLeft="20dp">
                <ImageView android:src="@drawable/batteryadmin"
android:layout width="wrap content" android:layout height="wrap content"
android:layout alignParentTop="true"/>
                <TextView android:text="节电管理"
android:layout width="wrap content" android:layout height="wrap content"
android:layout marginLeft="15dp" android:gravity="center horizontal"/>
            </LinearLayout>
            <LinearLayout android:id="@+id/settings"
android:layout height="wrap content" android:layout width="wrap content"
android:orientation="vertical" android:layout marginLeft="20dp">
                <ImageView android:src="@drawable/settings"
android:layout width="wrap content" android:layout height="wrap content"
android:layout alignParentTop="true"/>
                <TextView android:text="优化设置"
android:layout width="wrap content" android:layout height="wrap content"
android:layout marginLeft="15dp" android:gravity="center horizontal"/>
            </LinearLayout>
        </TableRow>
    </TableLayout>
</RelativeLayout>
```

在上述代码中，也进行了层级优化，布局后的界面效果如图 10-5 所示。启动 SDK 目录下的 tools 文件夹中的 hierarchyviewer.bat，可以查看当前 UI 的结构视图，如图 10-7

所示。

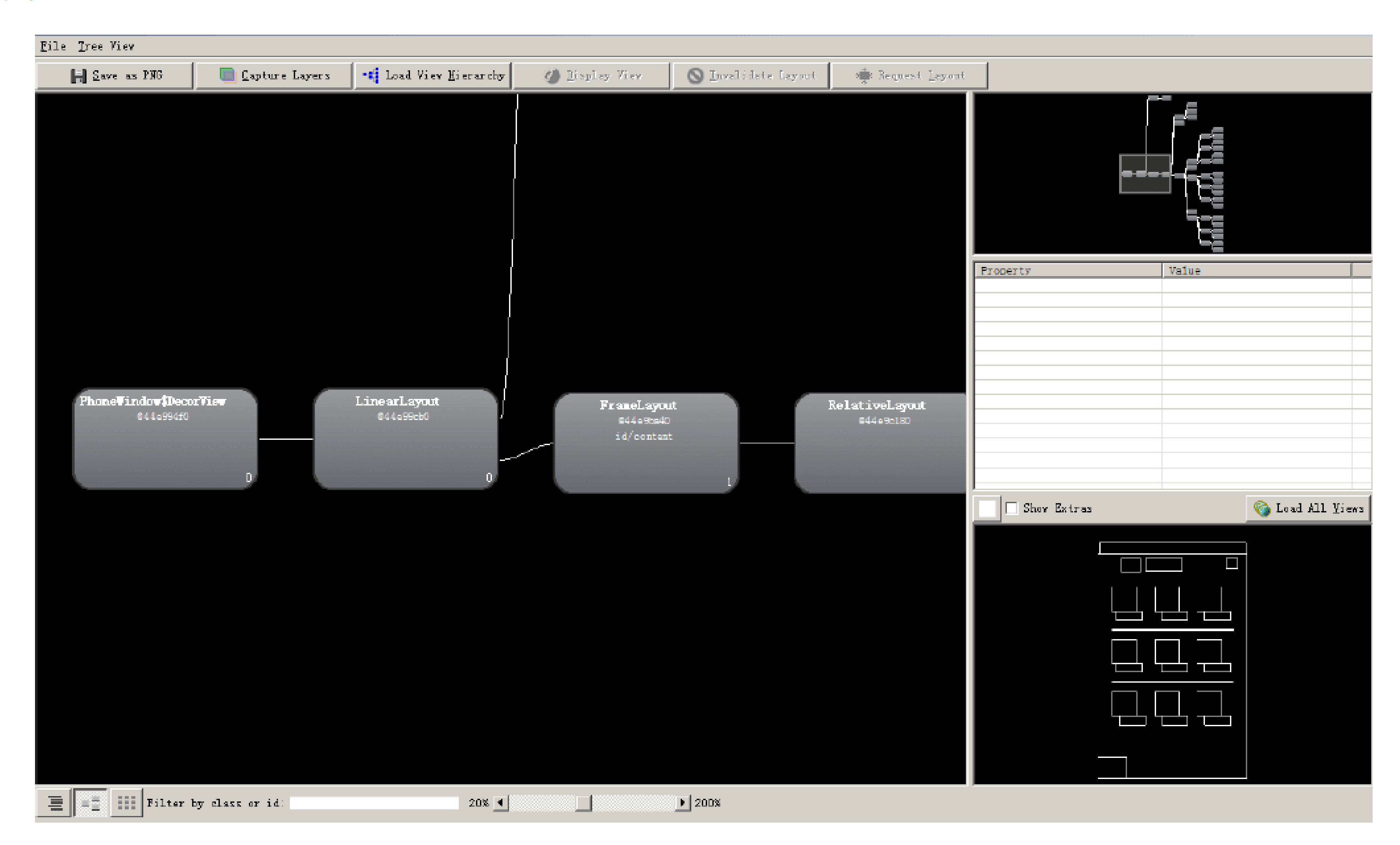


图 10-7　UI 结构视图

(3) 编写文件 file_category.xml，此文件的功能是实现文件管理模块的主界面效果。具体代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout width="fill parent"
android:layout_height="fill_parent"
android:background="@drawable/list bg" android:paddingTop="30dp"
android:paddingLeft="10dp" android:paddingRight="10dp">
    <LinearLayout android:orientation="vertical"
android:layout width="fill parent" android:layout height="fill parent">
        <LinearLayout android:orientation="horizontal"
android:layout width="fill parent" android:layout height="wrap content">
            <LinearLayout android:orientation="vertical"
android:layout width="wrap content" android:layout height="wrap content"
android:layout margin="20.0dip" android:layout weight="0.5">
                <RelativeLayout android:layout width="fill parent"
android:layout_height="wrap_content">
                    <ImageView android:src="@drawable/file category pic"
android:layout width="72.0dip" android:layout height="72.0dip"
android:baselineAlignBottom="true"
android:layout centerHorizontal="true"/>
                </RelativeLayout>
                <TextView android:text="图片浏览"
android:layout width="fill parent" android:layout height="wrap content"
android:gravity="center" android:textColor="@android:color/black"
android:textSize="16.0sp"/>
```



```
            </LinearLayout>
            <ImageView android:src="@drawable/main divider"
android:layout width="2.0dip" android:layout height="fill parent"
android:scaleType="fitXY"/>
            <LinearLayout android:orientation="vertical"
android:layout width="wrap content" android:layout height="wrap content"
android:layout margin="20.0dip" android:layout weight="0.5">
                <RelativeLayout android:layout width="fill parent"
android:layout height="wrap content">
                    <ImageView
android:src="@drawable/file category music"
android:layout width="72.0dip" android:layout height="72.0dip"
android:baselineAlignBottom="true"
android:layout centerHorizontal="true"/>
                </RelativeLayout>
                <TextView android:text="音乐浏览"
android:layout width="fill parent" android:layout height="wrap content"
android:gravity="center" android:textColor="@android:color/black"
android:textSize="16.0sp"/>
            </LinearLayout>
        </LinearLayout>
        <ImageView android:src="@drawable/main divider"
android:layout width="fill parent" android:layout height="2.0dip"
android:scaleType="fitXY"/>
        <LinearLayout android:orientation="horizontal"
android:layout width="fill parent" android:layout height="wrap content">
            <LinearLayout android:orientation="vertical"
android:layout width="wrap content" android:layout height="wrap content"
android:layout margin="20.0dip" android:layout weight="0.5">
                <RelativeLayout android:layout width="fill parent"
android:layout height="wrap content">
                    <ImageView
android:src="@drawable/file category movie"
android:layout width="72.0dip" android:layout height="72.0dip"
android:baselineAlignBottom="true"
android:layout centerHorizontal="true"/>
                </RelativeLayout>
                <TextView android:text="视频浏览"
android:layout width="fill parent" android:layout height="wrap content"
android:gravity="center" android:textColor="@android:color/black"
android:textSize="16.0sp"/>
            </LinearLayout>
            <ImageView android:src="@drawable/main divider"
android:layout width="2.0dip" android:layout height="fill parent"
android:scaleType="fitXY"/>
            <LinearLayout android:orientation="vertical"
android:layout width="wrap content" android:layout height="wrap content"
android:layout margin="20.0dip" android:layout weight="0.5">
                <RelativeLayout android:layout width="fill parent"
android:layout height="wrap content">
                    <ImageView android:src="@drawable/file_category_text"
```

```
android:layout width="72.0dip" android:layout height="72.0dip"
android:baselineAlignBottom="true"
android:layout centerHorizontal="true"/>
                </RelativeLayout>
                <TextView android:text="文档浏览"
android:layout width="fill parent" android:layout height="wrap content"
android:gravity="center" android:textColor="@android:color/black"
android:textSize="16.0sp"/>
            </LinearLayout>
        </LinearLayout>
    </LinearLayout>
</RelativeLayout>
```

笔者对上述文件也专门进行了 UI 优化，读者同样可以利用 hierarchyviewer.bat 查看结构视图。上述代码的 UI 效果如图 10-8 所示。

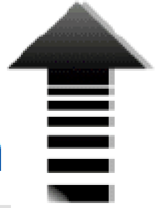


图 10-8　文件管理模块的主界面

10.4　编写主界面程序

图 10-5 所示的 UI 界面设计完毕后，在本节开始讲解此界面的程序文件。和此界面对应的程序文件是 NineGridActivity.java，此文件的功能是获取用户的触发事件，根据用户触摸的图标来到对应的界面。文件 NineGridActivity.java 的实现代码如下。

```
package com.process.ui.main;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.LinearLayout;
import com.process.R;
```



```
import com.process.ui.file.FileTabActivity;
import com.process.ui.task.TaskTabActivity;
public class NineGridActivity extends Activity implements OnClickListener {
    private LinearLayout checkhealth, proadmin, netadmin, install, adminpro,
            clear, fileadmin, batteryadmin, settings;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.nine grid);
        setUpViews();
        setListeners();
    }
    private void setUpViews() {
        checkhealth = (LinearLayout) findViewById(R.id.checkhealth);
        proadmin = (LinearLayout) findViewById(R.id.proadmin);
        netadmin = (LinearLayout) findViewById(R.id.netadmin);
        install = (LinearLayout) findViewById(R.id.install);
        adminpro = (LinearLayout) findViewById(R.id.adminpro);
        clear = (LinearLayout) findViewById(R.id.clear);
        fileadmin = (LinearLayout) findViewById(R.id.fileadmin);
        batteryadmin = (LinearLayout) findViewById(R.id.batteryadmin);
        settings = (LinearLayout) findViewById(R.id.settings);
    }
    private void setListeners() {
        checkhealth.setOnClickListener(this);
        proadmin.setOnClickListener(this);
        netadmin.setOnClickListener(this);
        install.setOnClickListener(this);
        adminpro.setOnClickListener(this);
        clear.setOnClickListener(this);
        fileadmin.setOnClickListener(this);
        batteryadmin.setOnClickListener(this);
        settings.setOnClickListener(this);
    }
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.checkhealth: {
        }
            break;
        case R.id.proadmin: {
        }
            break;
        case R.id.netadmin: {
        }
            break;
        case R.id.install: {
        }
            break;
        case R.id.adminpro: {
            Intent intent= new Intent(NineGridActivity.this,
```

```
TaskTabActivity.class);
                startActivity(intent);
            }
                break;
            case R.id.clear: {
            }
                break;
            case R.id.fileadmin: {
                Intent intent= new Intent(NineGridActivity.this,
FileTabActivity.class);
                startActivity(intent);
            }
                break;
            case R.id.batteryadmin: {
            }
                break;
            case R.id.settings: {
            }
                break;
            default:
                break;
            }
        }
}
```

10.5　进程管理模式模块

在进程管理模式中，总体设置和进程管理有关的变量，这些变量供本项目的其他模块使用。另外，还获取了每个进程所占用的内存信息和 CPU 信息，如图 10-9 所示。



图 10-9　进程列表



本节将详细讲解使用 Java 语言编写进程管理模式模块的具体流程。

10.5.1　基础状态文件

编写基础文件 BasicProgramUtil.java，在此文件中分别设置了图标、进程名、文件名和 CPU 模式变量，供其他模块的程序使用。文件 BasicProgramUtil.java 的实现代码如下。

```
package com.process.model;
import java.io.Serializable;
import android.graphics.drawable.Drawable;
public class BasicProgramUtil implements Serializable{
   /*
    * 定义应用程序的简要信息部分
    */
   private Drawable icon;                           // 程序图标
   private String programName;                      // 程序名称
   private String processName;
   private String cpuMemString;

   public BasicProgramUtil() {
      icon = null;
      programName = "";
      processName = "";
      cpuMemString = "";
   }
    public Drawable getIcon() {
        return icon;
    }
    public void setIcon(Drawable icon) {
        this.icon = icon;
    }
    public String getProgramName() {
        return programName;
    }
    public void setProgramName(String programName) {
        this.programName = programName;
    }
    public String getProcessName() {
        return processName;
    }
    public void setProcessName(String processName) {
        this.processName = processName;
    }
    public String getCpuMemString() {
        return cpuMemString;
    }
    public void setCpuMemString(String cpuMemString) {
        this.cpuMemString = cpuMemString;
    }
}
```

10.5.2　CPU 和内存使用信息

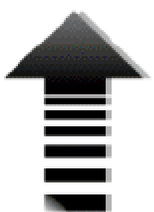
编写文件 CpuAndMemoryModel.java，通过此文件获取了每个进程占用的 CPU 和内存的信息，定义了和进程有关的构造函数。文件 CpuAndMemoryModel.java 的主要代码如下。

```
public class CpuAndMemoryModel implements Serializable {
    private String programName;
    private String processName;
    private String cpuString;
    private String memoryString;
    public String getProgramName() {
        return programName;
    }
    public void setProgramName(String programName) {
        this.programName = programName;
    }
    public String getProcessName() {
        return processName;
    }
    public void setProcessName(String processName) {
        this.processName = processName;
    }
    public String getCpuString() {
        return cpuString;
    }
    public void setCpuString(String cpuString) {
        this.cpuString = cpuString;
    }
    public String getMemoryString() {
        return memoryString;
    }
    public void setMemoryString(String memoryString) {
        this.memoryString = memoryString;
    }
}
```

10.5.3　进程详情

编写文件 DetailProgramUtil.java，其功能是设置了和进程有关的各个变量，显示某个进程的详细信息。文件 DetailProgramUtil.java 的主要代码如下。

```
public class DetailProgramUtil implements Serializable{
   private static final long serialVersionUID = 1L;
   /*
    * 定义应用程序的扩展信息部分
    */
   private int pid;
   private String processName;              // 程序运行的进程名
```



```
private String companyName;              // 公司名称
private int versionCode;                 // 版本代号
private String versionName;              // 版本名称

private String dataDir;                  // 程序的数据目录
private String sourceDir;                // 程序包的源目录
private String firstInstallTime;         // 第一次安装的时间
private String lastUpdateTime;           // 最近的更新时间

private String userPermissions;          // 应用程序的权限
private String activities;               // 应用程序包含的 Activities
private String services;                 // 应用程序包含的服务
// android.content.pm.PackageState 类的包信息
// 此处只是安装包的信息
private String codeSize;
private long dataSize;
private long cacheSize;
private long externalDataSize;
private long externalCacheSize;
private long externalMediaSize;
private long externalObbSize;
public DetailProgramUtil() {
    pid = 0;
    processName = "";
    companyName = "";
    versionCode = 0;
    versionName = "";
    dataDir = "";
    sourceDir = "";
    firstInstallTime = "";
    lastUpdateTime = "";
    userPermissions = "";
    activities = "";
    services = "";

    initPackageSize();
}
private void initPackageSize() {
    codeSize = "0.00";
    dataSize = 0;
    cacheSize = 0;
    externalCacheSize = 0;
    externalDataSize = 0;
    externalMediaSize = 0;
    externalObbSize = 0;
}
public int getPid() {
    return pid;
}
public void setPid(int pid) {
```

```
        this.pid = pid;
    }
    public int getVersionCode() {
        return versionCode;
    }
    public void setVersionCode(int versionCode) {
        this.versionCode = versionCode;
    }
    public String getVersionName() {
        return versionName;
    }
    public void setVersionName(String versionName) {
        this.versionName = versionName;
    }
    public String getCompanyName() {
        return companyName;
    }
    public void setCompanyName(String companyString) {
        this.companyName = companyString;
    }
    public String getFirstInstallTime() {
        if (firstInstallTime == null || firstInstallTime.length() <= 0) {
            firstInstallTime = "null";
        }
        return firstInstallTime;
    }
    public void setFirstInstallTime(long firstInstallTime) {
        this.firstInstallTime = DateFormat.format(
                "yyyy-MM-dd", firstInstallTime).toString();
    }
    public String getLastUpdateTime() {
        if (lastUpdateTime == null || lastUpdateTime.length() <= 0) {
            lastUpdateTime = "null";
        }
        return lastUpdateTime;
    }
    public void setLastUpdateTime(long lastUpdateTime) {
        this.lastUpdateTime = DateFormat.format(
                "yyyy-MM-dd", lastUpdateTime).toString();
    }
    public String getActivities() {
        if (activities == null || activities.length() <= 0) {
            activities = "null";
        }
        return activities;
    }
    public void setActivities(ActivityInfo[] activities) {
        this.activities = Array2String(activities);
    }
    public String getUserPermissions() {
        if (userPermissions == null || userPermissions.length() <= 0) {
```

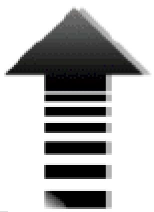


```
        userPermissions = "null";
    }
    return userPermissions;
}
public void setUserPermissions(String[] userPermissions) {
    this.userPermissions = Array2String(userPermissions);
}
public String getServices() {
    if (services == null || services.length() <= 0) {
        services = "null";
    }
    return services;
}
public void setServices(ServiceInfo[] services) {
    this.services = Array2String(services);
}
public String getProcessName() {
    if (processName == null || processName.length() <= 0) {
        processName = "null";
    }
    return processName;
}
public void setProcessName(String processName) {
    this.processName = processName;
}
public String getDataDir() {
    if (dataDir == null || dataDir.length() <= 0) {
        dataDir = "null";
    }
    return dataDir;
}
public void setDataDir(String dataDir) {
    this.dataDir = dataDir;
}
public String getSourceDir() {
    if (sourceDir == null || sourceDir.length() <= 0) {
        sourceDir = "null";
    }
    return sourceDir;
}
public void setSourceDir(String sourceDir) {
    this.sourceDir = sourceDir;
}

/*
 * 三个重载方法，参数不同，调用不同的方法，用于将 object 数组转化成要求的字符串
 */
// 用户权限信息
public String Array2String(String[] array) {

    String resultString = "";
```

```
    if (array != null && array.length > 0) {
        resultString = "";
        for (int i = 0; i < array.length; i++) {
            resultString += array[i];
            if (i < (array.length - 1)) {
                resultString += "\n";
            }
        }
    }
    return resultString;
}
// 服务信息
public String Array2String(ServiceInfo[] array) {
    String resultString = "";
    if (array != null && array.length > 0) {
        resultString = "";
        for (int i = 0; i < array.length; i++) {
            if (array[i].name == null) {
                continue;
            }
            resultString += array[i].name.toString();
            if (i < (array.length - 1)) {
                resultString += "\n";
            }
        }
    }
    return resultString;
}
// 活动信息
public String Array2String(ActivityInfo[] array) {
    String resultString = "";
    if (array != null && array.length > 0) {
        resultString = "";
        for (int i = 0; i < array.length; i++) {
            if (array[i].name == null) {
                continue;
            }
            resultString += array[i].name.toString();
            if (i < (array.length - 1)) {
                resultString += "\n";
            }
        }
    }
    return resultString;
}
public String getCodeSize() {
    return codeSize;
}
public void setCodeSize(long codeSize) {
    DecimalFormat df = new DecimalFormat("###.00");
    this.codeSize = df.format((double)(codeSize/1024.0));
```



```
    }
    public long getDataSize() {
        return dataSize;
    }
    public void setDataSize(long dataSize) {
        this.dataSize = dataSize;
    }
    public long getCacheSize() {
        return cacheSize;
    }
    public void setCacheSize(long cacheSize) {
        this.cacheSize = cacheSize;
    }
    public long getExternalDataSize() {
        return externalDataSize;
    }
    public void setExternalDataSize(long externalDataSize) {
        this.externalDataSize = externalDataSize;
    }
    public long getExternalCacheSize() {
        return externalCacheSize;
    }
    public void setExternalCacheSize(long externalCacheSize) {
        this.externalCacheSize = externalCacheSize;
    }
    public long getExternalMediaSize() {
        return externalMediaSize;
    }
    public void setExternalMediaSize(long externalMediaSize) {
        this.externalMediaSize = externalMediaSize;
    }
    public long getExternalObbSize() {
        return externalObbSize;
    }
    public void setExternalObbSize(long externalObbSize) {
        this.externalObbSize = externalObbSize;
    }

    public String getPackageSize() {
        String resultString = "";
        resultString = "Code Size: " + codeSize + "KB\n"
                   + "Data Size: " + dataSize + "KB\n"
                   + "Cache Size: " + cacheSize + "KB\n"
                   + "External Data Size: " + externalDataSize + "KB\n"
                   + "External Cache Size: " + externalCacheSize + "KB\n"
                   + "External Media Size: " + externalMediaSize + "KB\n"
                   + "External Obb Size: " + externalObbSize + "KB";
        return resultString;
    }
}
```

10.6 进程视图模块

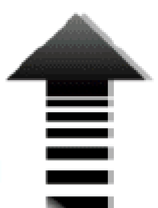
本模块的功能是，在进程主界面显示当前手机的进程信息，并获取每一个进程信息对象，显示此进程的详细信息。本节将详细讲解使用 Java 语言编写进程视图模块的具体流程。

10.6.1 进程主视图

编写文件 MainActivity.java，功能是以列表的样式显示手机内的进程信息，分别定义了进程、任务、服务、图标和文件变量。文件 MainActivity.java 的主要代码如下。

```
public class MainActivity extends TabActivity {
    private TabHost tabHost;
    private RadioGroup mainbtGroup;
    private static final String PROCESS = "进程";
    private static final String TASK = "任务";
    private static final String SERVICE = "服务";
    private static final String CHART = "图标";
    private static final String FILE = "文件";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.tabhost);
        tabHost = this.getTabHost();
        TabSpec tabSpec1 = tabHost.newTabSpec(PROCESS).setIndicator(PROCESS);
        tabSpec1.setContent(new Intent(this, ProcessActivity.class));
        TabSpec tabSpec2 = tabHost.newTabSpec(TASK).setIndicator(TASK);
        tabSpec2.setContent(new Intent(this, TaskActivity.class));
        TabSpec tabSpec3 = tabHost.newTabSpec(SERVICE).setIndicator(SERVICE);
        tabSpec3.setContent(new Intent(this, ServiceActivity.class));
        TabSpec tabSpec4 = tabHost.newTabSpec(CHART).setIndicator(CHART);
        tabSpec4.setContent(new Intent(this, ChartActivity.class));
        TabSpec tabSpec5 = tabHost.newTabSpec(FILE).setIndicator(FILE);
        tabSpec5.setContent(new Intent(this, FileActivity.class));

        tabHost.addTab(tabSpec1);
        tabHost.addTab(tabSpec2);
        tabHost.addTab(tabSpec3);
        tabHost.addTab(tabSpec4);
        tabHost.addTab(tabSpec5);
        mainbtGroup = (RadioGroup) this.findViewById(R.id.main_radio);
        mainbtGroup.setOnCheckedChangeListener(new OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(RadioGroup group, int checkedId) {
                switch (checkedId) {
                    case R.id.btn1:
                        tabHost.setCurrentTabByTag(PROCESS);
```



```
                        break;
                    case R.id.btn2:
                        tabHost.setCurrentTabByTag(TASK);
                        break;
                    case R.id.btn3:
                        tabHost.setCurrentTabByTag(SERVICE);
                        break;
                    case R.id.btn4:
                        tabHost.setCurrentTabByTag(CHART);
                        break;
                    case R.id.btn5:
                        tabHost.setCurrentTabByTag(FILE);
                        break;
                }
            }
        });
    }
}
```

10.6.2　进程视图

编写文件 TaskActivity.java，此文件比较简单，功能是分别定义方法 onResume()和 oncreate()，设置进入不同的视图模式。文件 TaskActivity.java 的主要代码如下。

```
package com.process.ui;
import android.app.Activity;
import android.app.ListActivity;
import android.os.Bundle;
import android.util.Log;
public class TaskActivity extends ListActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d("TaskActivity", "进入 oncreate 方法");
    }

    @Override
    protected void onResume() {
        super.onResume();
        Log.d("TaskActivity", "进入 onResume 方法");
    }
}
```

10.6.3　获取进程信息

编写文件 DetailActivity.java，功能是根据某个进程的名字获取应用程序的 ApplicationInfo 对象，然后显示出此进程的详细信息。文件 DetailActivity.java 的主要代码如下。

```
public class DetailActivity extends Activity {
    private PackageManager packageManager;
    private ProcessMemoryUtil processMemoryUtil;
    private PackageUtil packageUtil;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        packageUtil = new PackageUtil(DetailActivity.this);
        Intent intent = getIntent();
        Bundle bundle = intent.getExtras();
        String procNameString = bundle.getString("procNameString");
        TextView tv = new TextView(DetailActivity.this);
        tv.setText(procNameString);
        setContentView(tv);
    }
    public DetailProgramUtil buildProgramUtilComplexInfo(String procNameString) {
       DetailProgramUtil complexProgramUtil = new DetailProgramUtil();
       // 根据进程名，获取应用程序的 ApplicationInfo 对象
       ApplicationInfo tempAppInfo =
           packageUtil.getApplicationInfo(procNameString);
       if (tempAppInfo == null) {
          return null;
       }
       PackageInfo tempPkgInfo = null;
       try {
          tempPkgInfo = packageManager.getPackageInfo(
                 tempAppInfo.packageName,
                 PackageManager.GET UNINSTALLED PACKAGES |
                 PackageManager.GET ACTIVITIES
                 | PackageManager.GET SERVICES |
                 PackageManager.GET PERMISSIONS);
       } catch (NameNotFoundException e) {
          e.printStackTrace();
       }
       if (tempPkgInfo == null) {
          return null;
       }
       complexProgramUtil.setProcessName(procNameString);
       complexProgramUtil.setCompanyName(getString(R.string.no use));
       complexProgramUtil.setVersionName(tempPkgInfo.versionName);
       complexProgramUtil.setVersionCode(tempPkgInfo.versionCode);
       complexProgramUtil.setDataDir(tempAppInfo.dataDir);
       complexProgramUtil.setSourceDir(tempAppInfo.sourceDir);
       // 以下注释部分的信息暂时获取不到
//       complexProgramUtil.setFirstInstallTime(tempPkgInfo.firstInstallTime);
//       complexProgramUtil.setLastUpdateTime(tempPkgInfo.lastUpdateTime);
//       complexProgramUtil.setCodeSize(packageStats.codeSize);
//       complexProgramUtil.setDataSize(packageStats.dataSize);
//       complexProgramUtil.setCacheSize(packageStats.cacheSize);
//       complexProgramUtil.setExternalDataSize(0);
//       complexProgramUtil.setExternalCacheSize(0);
```

```
//       complexProgramUtil.setExternalMediaSize(0);
//       complexProgramUtil.setExternalObbSize(0);
      // 获取以下三个信息，需要为 PackageManager 进行授权
(packageManager.getPackageInfo()方法)
      complexProgramUtil.setUserPermissions(tempPkgInfo.requestedPermissions);
      complexProgramUtil.setServices(tempPkgInfo.services);
      complexProgramUtil.setActivities(tempPkgInfo.activities);
      return complexProgramUtil;
   }
}
```

10.7　进程类别模块

本模块的功能是，将进程分为了两类：运行中程序和运行中服务，并且设置了进度条效果显示系统内的进程。本节将详细讲解使用 Java 语言编写进程类别模块的具体流程。

10.7.1　加载进程

编写文件 ProcessActivity.java，功能是以进度条的样式加载当前手机中运行的进程。文件 ProcessActivity.java 的主要代码如下。

```
public class ProcessActivity extends ListActivity implements
OnItemLongClickListener,OnItemClickListener{
    private PackageManager packageManager;
    private ProgressDialog pd;
    private Handler handler;
    private List<BasicProgramUtil> list = null;
    private PackageUtil packageUtil;
    private ProcessMemoryUtil processMemoryUtil;
    private ListView listView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.process);
        listView = getListView();
        listView.setOnItemLongClickListener(this);//为 listView 添加
        listView.setOnItemClickListener(this);
        packageUtil = new PackageUtil(ProcessActivity.this);
        processMemoryUtil = new ProcessMemoryUtil();
        packageManager = getPackageManager();
        pd = new ProgressDialog(ProcessActivity.this);// 生成一个进度条
        pd.setProgressStyle(ProgressDialog.STYLE_SPINNER);
        pd.setTitle(getString(R.string.progress_tips_title));
        pd.setMessage(getString(R.string.progress_tips_content));
        handler = new RefreshHandler();
        pd.show();
        RefreshThread thread = new RefreshThread();
```




```
        thread.start();// 耗时操作，需要开启一个线程
    }

    @Override
    protected void onResume() {
        super.onResume();
    }
    class RefreshHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            refreshListItems();
            setTitle("软件信息,有" + list.size() + "个进程在运行.");
            pd.dismiss();// 关闭进度条
        }
    }
    class RefreshThread extends Thread {
        @Override
        public void run() {
            getRunningAppProcesses();
            Message msg = handler.obtainMessage();
            handler.sendMessage(msg);
        }
    }
    private void refreshListItems() {
        list = getRunningAppProcesses();
        MyAdapter adapter = new MyAdapter(ProcessActivity.this, list);
        listView.setAdapter(adapter);
    }

    private List<BasicProgramUtil> getRunningAppProcesses() {
        ActivityManager activityManager = (ActivityManager)
            getSystemService(ACTIVITY_SERVICE);
        List<RunningAppProcessInfo> procList =
            activityManager.getRunningAppProcesses();
        List list = new ArrayList<BasicProgramUtil>();
        for (Iterator<RunningAppProcessInfo> iterator =
            procList.iterator(); iterator
                .hasNext();) {
            RunningAppProcessInfo procInfo = iterator.next();
            BasicProgramUtil basicProgramUtil =
                buildProgramUtilSimpleInfo(procInfo.pid, procInfo.processName);
            list.add(basicProgramUtil);
        }
        return list;
    }
    private void returnToHome() {
       Intent home = new Intent(Intent.ACTION_MAIN);
       home.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
       home.addCategory(Intent.CATEGORY_HOME);
       startActivity(home);
   }
```



```
    public BasicProgramUtil buildProgramUtilSimpleInfo(int procId,
        String procNameString) {
        BasicProgramUtil programUtil = new BasicProgramUtil();
        programUtil.setProcessName(procNameString);
        // 根据进程名，获取应用程序的 ApplicationInfo 对象
        ApplicationInfo tempAppInfo =
            packageUtil.getApplicationInfo(procNameString);
        if (tempAppInfo != null) {
            // 为进程加载图标和程序名称
            programUtil.setIcon(tempAppInfo.loadIcon(packageManager));
            programUtil.setProgramName(tempAppInfo.loadLabel
(packageManager).toString());
        }
        else {
            // 如果获取失败，则使用默认的图标和程序名
programUtil.setIcon(getApplicationContext().getResources().getDrawable(R
.drawable.unknown));
            programUtil.setProgramName(procNameString);
        }
        String str = processMemoryUtil.getMemInfoByPid(procId);
        programUtil.setCpuMemString(str);
        return programUtil;
    }
    class MyAdapter extends BaseAdapter{
        private Context context;
        private List<BasicProgramUtil> list;
        private LayoutInflater inflater;
        public MyAdapter(Context context,List<BasicProgramUtil> list){
            super();
            this.context = context;
            this.list = list;
            this.inflater = LayoutInflater.from(context);
        }
        @Override
        public int getCount() {
            return list.size();
        }
        @Override
        public Object getItem(int position) {
            return list.get(position);
        }
        @Override
        public long getItemId(int position) {
            return position;
        }
        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
            BasicProgramUtil bp = list.get(position);
            View v = convertView;
            if(v==null){
                v = inflater.inflate(R.layout.proc_list_item, null);
```




```
                ViewHolder viewHolder = new ViewHolder();
                viewHolder.img = (ImageView)v.findViewById(R.id.icon);
                viewHolder.tv1 =  (TextView)v.findViewById(R.id.programName);
               //viewHolder.tv2 =  (TextView)v.findViewById(R.id.processName);
                viewHolder.tv3 =  (TextView)v.findViewById(R.id.cpuMemString);
                v.setTag(viewHolder);
            }
            ViewHolder viewHolder = (ViewHolder)v.getTag();
            viewHolder.img.setBackgroundDrawable(bp.getIcon());
            viewHolder.tv1.setText(bp.getProgramName());
            //viewHolder.tv2.setText(bp.getProcessName());
            viewHolder.tv3.setText(bp.getCpuMemString());
            return v;
        }
    }
    static class ViewHolder{
        private ImageView img;
        private TextView tv1;
        //private TextView tv2;
        private TextView tv3;
    }
    @Override
    public boolean onItemLongClick(AdapterView<?> arg0, View arg1, int
        position,long arg3) {
        return false;
    }
    @Override
   public void onItemClick(AdapterView<?> arg0, View arg1, int position,
        long arg3) {
        final BasicProgramUtil bsu = list.get(position);
        final Intent intent = new
            Intent(ProcessActivity.this,DetailActivity.class);
        Bundle bundle = new Bundle();
        bundle.putString("procNameString", bsu.getProcessName());
        intent.putExtras(bundle);

        AlertDialog.Builder builder = new
            AlertDialog.Builder(ProcessActivity.this);
        builder.setTitle("查看详情 or 结束此进程");
        builder.setIcon(R.drawable.question);
        builder.setPositiveButton("详情", new
            DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                startActivity(intent);//跳往程序详情显示页面
            }
        });
        builder.setNegativeButton("结束此进程", new
            DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
```

```
                //结束此进程
            }
        });
        builder.show();
    }
}
```

10.7.2　后台加载设置

编写文件 ServiceActivity.java，功能是根据用户的选择加载执行不同的方法，这样可以分别进入运行中程序和运行中服务模式。文件 ServiceActivity.java 的主要代码如下。

```
public class ServiceActivity extends ListActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d("ServiceActivity", "进入 onCreate 方法");
    }
    @Override
    protected void onResume() {
        super.onResume();
        Log.d("ServiceActivity", "进入 onResume 方法");
    }
}
```

10.7.3　加载显示

编写文件 TaskTabActivity.java，功能是分别定义两个 View 对象：view1 和 view2，根据用户的需要进入不同的视图界面。其中 view1 表示运行中的程序视图，view2 表示运行中的服务视图。文件 TaskTabActivity.java 的主要代码如下。

```
public class TaskTabActivity extends TabActivity {
    private TabHost tabHost;
    private static final String RUNNINGPROGRAM = "运行中程序";
    private static final String RUNNINGSERVICE = "运行中服务";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.tabhost);
        tabHost = getTabHost();
        View view1 = View.inflate(TaskTabActivity.this, R.layout.tab, null);
        ((ImageView) view1.findViewById(R.id.tab imageview icon)).
            setImageResource(R.drawable.task tab1 icon);
        ((TextView) view1.findViewById(R.id.tab textview title)).setText
            (RUNNINGPROGRAM);
        TabHost.TabSpec spec1 = tabHost.newTabSpec(RUNNINGPROGRAM)
                .setIndicator(view1)
                .setContent(new Intent(this, ProcessActivity.class));
        tabHost.addTab(spec1);
```

```
        View view2 = View.inflate(TaskTabActivity.this, R.layout.tab, null);
        ((ImageView) view2.findViewById(R.id.tab imageview icon)).
            setImageResource(R.drawable.task tab2 icon);
        ((TextView) view2.findViewById(R.id.tab textview title)).setText
            (RUNNINGSERVICE);
        TabHost.TabSpec spec2 = tabHost.newTabSpec(RUNNINGPROGRAM)
                .setIndicator(view2)
                .setContent(new Intent(this, ServiceActivity.class));
        tabHost.addTab(spec2);
    }
}
```

10.8　文件管理模式模块

本模块的功能是，设置当前手机设备中的文件模式，我们可以将文件分为不同的类别，并快速打开相应类别的文件。本节将详细讲解使用 Java 语言编写文件管理模式模块的具体流程。

10.8.1　文件分类

编写文件 FileCategoryActivity.java，功能是实现文件的分类，将当前手机设备中的文件分为如下 4 种类型。

- 图片
- 音乐
- 视频
- 文档

文件 FileCategoryActivity.java 的实现代码如下。

```
package com.process.ui.file;
import com.process.R;
import android.app.Activity;
import android.os.Bundle;
public class FileCategoryActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.file category);
    }
}
```

10.8.2　加载进程

编写文件 FileActivity.java，功能是响应用户的选择，并根据选择显示左侧或右侧对应



目录中的子目录。文件 FileActivity.java 的主要代码如下。

```
public class FileActivity extends Activity{
    private ListView leftLV,rightLV;
    List<Map<String, Object>> leftList;
    List<Map<String, Object>> rightList;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.file);

        leftLV = (ListView)findViewById(R.id.leftLV);
        rightLV = (ListView)findViewById(R.id.rightLV);

        List<Map<String, Object>> fileList = new
            ArrayList<Map<String,Object>>();
        FileUtil.getParentPath(new File("/"), fileList);
        leftList = fileList;
        rightList = FileUtil.getSubDirAndFiles(new File("/"));

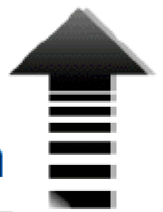
        setUpAdapter();//填充初始数据
        leftLV.setOnItemClickListener(new LeftItemListener());
        rightLV.setOnItemClickListener(new RightItemListener());
        rightLV.setOnItemLongClickListener(new
            rightLVItemLongClickListener());
    }
    private void setUpAdapter(){
        if(leftList!=null){
            SimpleAdapter leftAdapter = new SimpleAdapter(this, leftList,
                R.layout.file_left_item,
                    new String[] { "currentDirImage", "currentDirName"},
                        new int[] { R.id.currentDirImage,
                            R.id.currentDirName});
            leftLV.setAdapter(leftAdapter);
        }else{
            leftLV.setAdapter(null);
        }
        if(rightList!=null){
            SimpleAdapter rightAdapter = new SimpleAdapter(this,
                rightList, R.layout.file_right_item,
                   new String[] { "subDirImage", "subDirName"}, new int[]
                        { R.id.subDirImage,
                            R.id.subDirName});
            rightLV.setAdapter(rightAdapter);
        }else{
            rightLV.setAdapter(null);
            Toast.makeText(FileActivity.this, "空文件夹",
                Toast.LENGTH_SHORT).show();
        }
    }
```

```
class LeftItemListener implements OnItemClickListener{
    @Override
    public void onItemClick(AdapterView<?> arg0, View arg1, int position,
            long arg3) {
        Map<String, Object> map = leftList.get(position);
        String currentDirPath = (String)map.get("currentDirPath");
        File file = new File(currentDirPath);

        List<Map<String, Object>> list = new ArrayList<Map<String,
            Object>>();
        FileUtil.getParentPath(file, list);
        leftList = list;
        rightList = FileUtil.getSubDirAndFiles(file);
        setUpAdapter();//刷新
    }
}
class RightItemListener implements OnItemClickListener{
    @Override
   public void onItemClick(AdapterView<?> arg0, View arg1, int position,
            long arg3) {
        Map<String, Object> map = rightList.get(position);
        String subDirPath = (String)map.get("subDirPath");
        File file = new File(subDirPath);
        File parentFile = file.getParentFile();

        if(file.isDirectory()){//处理左边目录与右边目录以及右边文件的显示
           List<Map<String, Object>> list = new ArrayList<Map<String,
                Object>>();
            FileUtil.getParentPath(file, list);
            leftList = list;
            rightList = FileUtil.getSubDirAndFiles(file);
            setUpAdapter();//刷新

        }else{//如果点击的是文件，提示用户选择相应的程序打开此文件
            Toast.makeText(FileActivity.this, "你选择的是文件",
                Toast.LENGTH SHORT).show();
        }
    }
}
@Override
public boolean onCreateOptionsMenu(Menu menu){
    super.onCreateOptionsMenu(menu);
    MenuInflater menuInflater = getMenuInflater();
    menuInflater.inflate(R.menu.filemenu, menu);
    return true;
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case R.id.addfolder:{
        AlertDialog.Builder builder = new
```



```
            AlertDialog.Builder(FileActivity.this);
        builder.setTitle("输入名字");
        builder.setIcon(R.drawable.directory);
        builder.setCancelable(true);

        LayoutInflater inflater = LayoutInflater.from(FileActivity.this);
        View rootView = inflater.inflate
            (R.layout.input foldername dialog, null);
        final EditText et = (EditText)rootView.
            findViewById(R.id.foldername);

        builder.setView(rootView);

        builder.setPositiveButton("确定", new
            DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
        String foldername = et.getText().toString();
        Map<String, Object> map = leftList.get(0);

        File parentFile = new File((String)map.get("currentDirPath"));
        File newFolder = new File(parentFile,foldername);
        if(newFolder.mkdir()){
        rightList = FileUtil.getSubDirAndFiles(parentFile);
        setUpAdapter();//刷新
            Toast.makeText(FileActivity.this, "创建成功",
                Toast.LENGTH SHORT).show();
            }else{
            Toast.makeText(FileActivity.this, "重名文件夹",
                Toast.LENGTH SHORT).show();
                    }
                }
            });
            builder.show();
                }
        return true;
    case R.id.deletefolder:{
        Toast.makeText(FileActivity.this, "删除文件夹",
            Toast.LENGTH SHORT).show();
                }
        return true;
    default:
        return false;
    }
}
class rightLVItemLongClickListener implements OnItemLongClickListener{
    @Override
    public boolean onItemLongClick(AdapterView<?> arg0, View arg1,
            int position, long arg3) {
        final Map<String, Object> map = rightList.get(0);
```

```
            AlertDialog.Builder builder = new
                AlertDialog.Builder(FileActivity.this);
            builder.setTitle("你确定要删除吗?");
            builder.setIcon(R.drawable.question);
            builder.setPositiveButton("确定", new
                DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    File currentFile = new
                        File((String)map.get("subDirPath"));
                    if(currentFile.delete()){
                        rightList = FileUtil.getSubDirAndFiles
                            (currentFile.getParentFile());
                        setUpAdapter();//刷新
                        Toast.makeText(FileActivity.this, "删除成功",
                            Toast.LENGTH_SHORT).show();
                    }else{
                        Toast.makeText(FileActivity.this, "删除失败",
                            Toast.LENGTH_SHORT).show();
                    }
                }
            });
            builder.setNegativeButton("取消", new
                DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                }
            });
            builder.show();
            return false;
        }

    }
}
```

10.8.3 文件视图处理

除了将文件按照类别进行管理外，还可以根据树图模式进行管理。编写文件FileTabActivity.java，根据用户选择的选项卡显示不同的文件管理模式，此文件的主要代码如下。

```
public class FileTabActivity extends TabActivity {
    private TabHost tabHost;
    private static final String VIEWBYTYPE = "分类管理";
    private static final String TREEADMIN = "树图管理";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.tabhost);
        tabHost = getTabHost();
```

```
        View view1 = View.inflate(FileTabActivity.this, R.layout.tab, null);
        ((ImageView) view1.findViewById(R.id.tab imageview icon)).
            setImageResource(R.drawable.file tab1 icon);
        ((TextView) view1.findViewById(R.id.tab textview title)).
            setText(VIEWBYTYPE);
        TabHost.TabSpec spec1 = tabHost.newTabSpec(VIEWBYTYPE)
                .setIndicator(view1)
               .setContent(new Intent(this, FileCategoryActivity.class));
        tabHost.addTab(spec1);
        View view2 = View.inflate(FileTabActivity.this, R.layout.tab, null);
        ((ImageView) view2.findViewById(R.id.tab imageview icon)).
            setImageResource(R.drawable.file tab2 icon);
        ((TextView) view2.findViewById(R.id.tab textview title)).
            setText(TREEADMIN);
        TabHost.TabSpec spec2 = tabHost.newTabSpec(VIEWBYTYPE)
                .setIndicator(view2)
                .setContent(new Intent(this, FileActivity.class));
        tabHost.addTab(spec2);
    }
}
```

如果用户单击“分类管理”选项卡，则显示如图 10-8 所示的界面，如果单击“树图管理”选项卡，则显示如图 10-10 所示的界面。

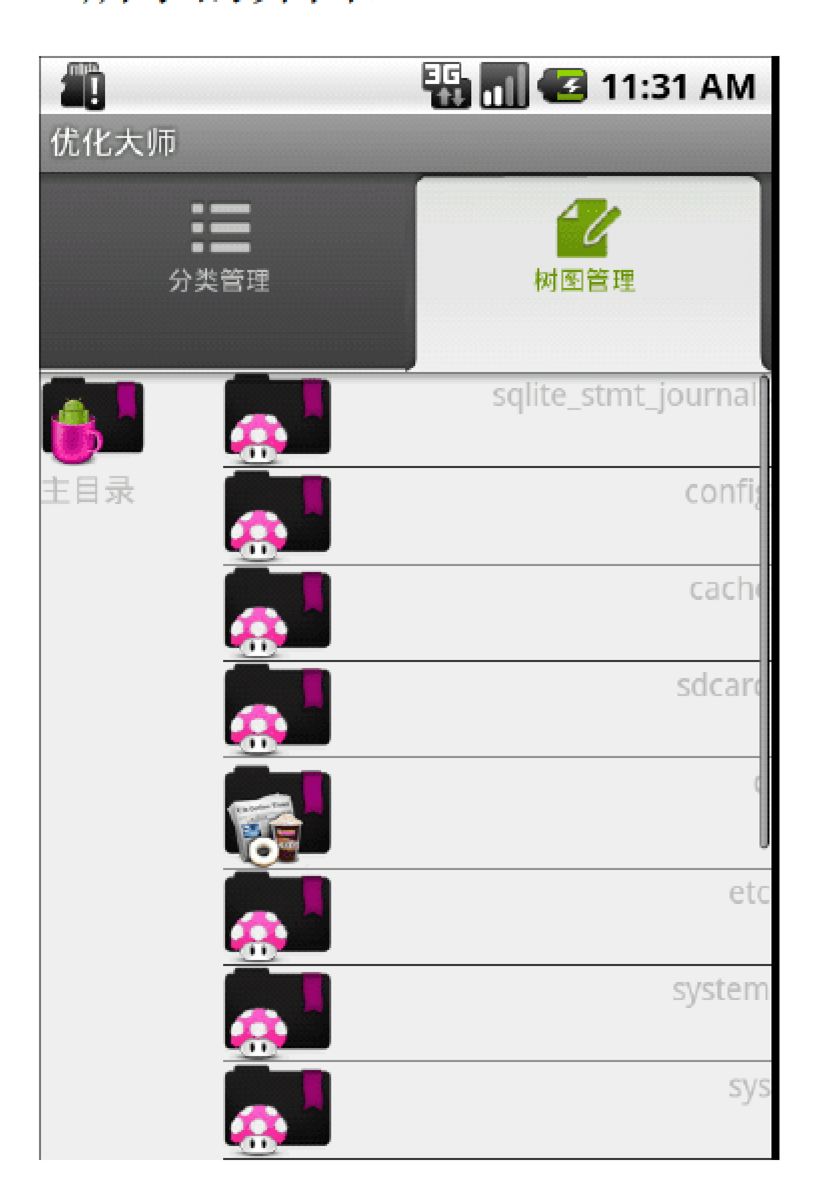


图 10-10　“树图管理”选项卡

10.9　文件管理模块

本模块的功能是，管理当前手机设备中的文件，查看某个目录下的子目录和孙目录等信息，并且对内存和 CPU 的使用信息进行了转换处理。本节将详细讲解使用 Java 语言编

写文件管理模块的具体流程。

10.9.1 文件夹

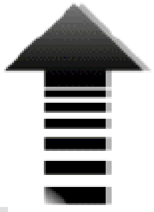
编写文件 PackageUtil.java，功能是通过包管理器检索所有的应用程序(包括卸载)与数据目录。此文件的主要代码如下。

```
public class PackageUtil {
    // ApplicationInfo 类，保存了普通应用程序的信息，主要是指 Manifest.xml 中
       application 标签中的信息
    private List<ApplicationInfo> allAppList;
    public PackageUtil(Context context) {
        // 通过包管理器，检索所有的应用程序(包括卸载)与数据目录
        PackageManager pm = context.getApplicationContext().getPackageManager();
        allAppList = pm.getInstalledApplications
            (PackageManager.GET UNINSTALLED PACKAGES);
        pm.getInstalledPackages(0);
    }
    /**
     * 通过一个程序名返回该程序的一个 ApplicationInfo 对象
     * @param name 程序名
     * @return ApplicationInfo
     */
    public ApplicationInfo getApplicationInfo(String appName) {
        if (appName == null) {
            return null;
        }
        for (ApplicationInfo appinfo : allAppList) {
            if (appName.equals(appinfo.processName)) {
                return appinfo;
            }
        }
        return null;
    }
}
```

10.9.2 显示文件信息

编写文件 FileUtil.java，功能获取并显示当前手机设备中的文件信息，包括文件名、文件格式和文件路径。文件 FileUtil.java 的主要代码如下。

```
public class FileUtil {
    public static void getParentPath(File file,List<Map<String,Object>> list){
        Map<String,Object> map = new HashMap<String, Object>();
    if(file.getName()==null||"".equals(file.getName())||"/".equals(file.
        getName())){
            map.put("currentDirName","主目录");
            map.put("currentDirImage",R.drawable.rootdir);
        }else if(file.getName().indexOf("sdcard")!=-1){
```



```
            map.put("currentDirName","sdcard");
            map.put("currentDirImage",R.drawable.sdcard);
        }else{
            map.put("currentDirName", file.getName());
            map.put("currentDirImage",R.drawable.directory);
        }
        map.put("currentDirPath", file.getAbsolutePath());
        list.add(map);
        if(file.getParentFile()!=null){
            getParentPath(file.getParentFile(),list);
        }
    }
    public static List<Map<String, Object>> getSubDirAndFiles(File pathFile){
        File[] files = pathFile.listFiles();
        if(files==null||files.length<1){
            return null;
        }
        List<Map<String, Object>> list = new ArrayList<Map<String,
            Object>>(files.length);
        for (File file : files){
            Map<String, Object> map = new HashMap<String, Object>();
            if(file.isDirectory()){
                map.put("subDirImage", R.drawable.directory);
            }else{
                String fileName = file.getName();
                if(fileName.indexOf("jpg")!=-1){
                    map.put("subDirImage", R.drawable.jpg);
                }else if(fileName.indexOf("txt")!=-1){
                    map.put("subDirImage", R.drawable.txt);
                }else if(fileName.indexOf("mp3")!=-1){
                    map.put("subDirImage", R.drawable.mp3);
                }else if(fileName.indexOf("avi")!=-1){
                    map.put("subDirImage", R.drawable.avi);
                }else if(fileName.indexOf("xls")!=-1){
                    map.put("subDirImage", R.drawable.excel);
                }else if(fileName.indexOf("mpeg")!=-1){
                    map.put("subDirImage", R.drawable.mpeg);
                }else if(fileName.indexOf("rar")!=-1){
                    map.put("subDirImage", R.drawable.rar);
                }else if(fileName.indexOf("tif")!=-1){
                    map.put("subDirImage", R.drawable.tif);
                }else if(fileName.indexOf("wav")!=-1){
                    map.put("subDirImage", R.drawable.wav);
                }else if(fileName.indexOf("wma")!=-1){
                    map.put("subDirImage", R.drawable.wma);
                }else if(fileName.indexOf("doc")!=-1){
                    map.put("subDirImage", R.drawable.word);
                }else if(fileName.indexOf("zip")!=-1){
                    map.put("subDirImage", R.drawable.zip);
                }else{
                    map.put("subDirImage", R.drawable.file);
```

```
                }
            }
            map.put("subDirName", file.getName());
            map.put("subDirPath", file.getPath());
            list.add(map);
        }
        return list;
    }
}
```

10.9.3 操作文件

编写文件 CMDExecute.java，功能是定义 ProcessBuilder 对象 builder，通过 CMD 方式操作一个文件。如果文件的路径为空，则关闭操作流。文件 CMDExecute.java 的主要代码如下。

```
public class CMDExecute {
    public synchronized String run(String[] cmd, String workdirectory)
            throws IOException {
        String result = "";
        try {
            ProcessBuilder builder = new ProcessBuilder(cmd);
            InputStream in = null;
            // 设置一个路径
            if (workdirectory != null) {
                builder.directory(new File(workdirectory));
                builder.redirectErrorStream(true);
                Process process = builder.start();
                in = process.getInputStream();
                byte[] re = new byte[1024];
                while (in.read(re) != -1)
                    result = result + new String(re);
            }
            if (in != null) {
                in.close();
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return result;
    }
}
```

10.9.4 获取进程的 CPU 和内存信息

编写文件 ProcessMemoryUtil.java，获取指定进程所占用的 CPU 信息和内存信息。在获取 CPU 的使用信息时，进行了百分比计算。文件 ProcessMemoryUtil.java 的主要实现代码如下。



```
public class ProcessMemoryUtil {

    private static final int INDEX_FIRST = -1;
    private static final int INDEX_PID = INDEX_FIRST + 1;
    private static final int INDEX_CPU = INDEX_FIRST + 2;
    private static final int INDEX_STAT = INDEX_FIRST + 3;
    private static final int INDEX_THR = INDEX_FIRST + 4;
    private static final int INDEX_VSS = INDEX_FIRST + 5;
    private static final int INDEX_RSS = INDEX_FIRST + 6;
    private static final int INDEX_PCY = INDEX_FIRST + 7;
    private static final int INDEX_UID = INDEX_FIRST + 8;
    private static final int INDEX_NAME = INDEX_FIRST + 9;
    private static final int Length_ProcStat = 9;

    private List<String[]> PMUList = null;

    public ProcessMemoryUtil() {
     initPMUtil();
    }

    private String getProcessRunningInfo() {
        Log.i("fetch process info", "start. . . . ");
        String result = null;
        CMDExecute cmdexe = new CMDExecute();
        try {
            String[] args = {"/system/bin/top", "-n", "1"};
            result = cmdexe.run(args, "/system/bin/");
        } catch (IOException ex) {
            Log.i("fetch process info", "ex=" + ex.toString());
        }
        return result;
    }

    private int parseProcessRunningInfo(String infoString) {
        String tempString = "";
        boolean bIsProcInfo = false;

        String[] rows = null;
        String[] columns = null;
        rows = infoString.split("[\n]+");        // 使用正则表达式分割字符串

        for (int i = 0; i < rows.length; i++) {
            tempString = rows[i];
            if (tempString.indexOf("PID") == -1) {
                if (bIsProcInfo == true) {
                    tempString = tempString.trim();
                    columns = tempString.split("[ ]+");
                    if (columns.length == Length_ProcStat) {
                        PMUList.add(columns);
                    }
```




```
            }
        } else {
            bIsProcInfo = true;
        }
    }

    return PMUList.size();
}

// 初始化所有进程的 CPU 和内存列表，用于检索每个进程的信息
public void initPMUtil() {
    PMUList = new ArrayList<String[]>();
    String resultString = getProcessRunningInfo();
    parseProcessRunningInfo(resultString);
}

// 根据进程名获取 CPU 和内存信息
public String getMemInfoByName(String procName) {
    String result = "";

    String tempString = "";
    for (Iterator<String[]> iterator = PMUList.iterator();
         iterator.hasNext();) {
        String[] item = (String[]) iterator.next();
        tempString = item[INDEX_NAME];
        if (tempString != null && tempString.equals(procName)) {
            result = "CPU:" + item[INDEX_CPU]
                    + "  内存:" + item[INDEX_RSS];
            break;
        }
    }
    return result;
}

// 根据进程 ID 获取 CPU 和内存信息
public String getMemInfoByPid(int pid) {
    String result = "";

    String tempPidString = "";
    int tempPid = 0;
    int count = PMUList.size();
    for (int i = 0; i < count; i++) {
        String[] item = PMUList.get(i);
        tempPidString = item[INDEX_PID];
        if (tempPidString == null) {
            continue;
        }
        tempPid = Integer.parseInt(tempPidString);
        if (tempPid == pid) {
            result = "CPU:" + item[INDEX_CPU]
                    + "  内存:" + item[INDEX_RSS];
```



```
                break;
            }
        }
        return result;
    }

    // 根据进程 ID 获取内存信息
    public String getMemorySizeByPid(int pid) {
        String result = "";

        String tempPidString = "";
        int tempPid = 0;
        int count = PMUList.size();
        for (int i = 0; i < count; i++) {
            String[] item = PMUList.get(i);
            tempPidString = item[INDEX PID];
            if (tempPidString == null) {
                continue;
            }
            tempPid = Integer.parseInt(tempPidString);
            if (tempPid == pid) {
              int size = item[INDEX RSS].length();
                result = item[INDEX RSS].substring(0,size-1);
                break;
            }
        }
        return result;
    }

    // 根据进程 ID 获取 CPU 信息
    public String getCPUSizeByPid(int pid) {
        String result = "";
        String tempPidString = "";
        int tempPid = 0;
        int count = PMUList.size();
        for (int i = 0; i < count; i++) {
            String[] item = PMUList.get(i);
            tempPidString = item[INDEX PID];
            if (tempPidString == null) {
                continue;
            }
            tempPid = Integer.parseInt(tempPidString);
            if (tempPid == pid) {
                result = item[INDEX CPU];
                break;
            }
        }
        return result;
    }
}
```

10.10 系 统 测 试

经过本章前面内容的讲解，一个基本的简化版 Android 优化系统开发完毕。在本节的内容中，开始进行具体测试。

执行后的主界面效果如图 10-11 所示。

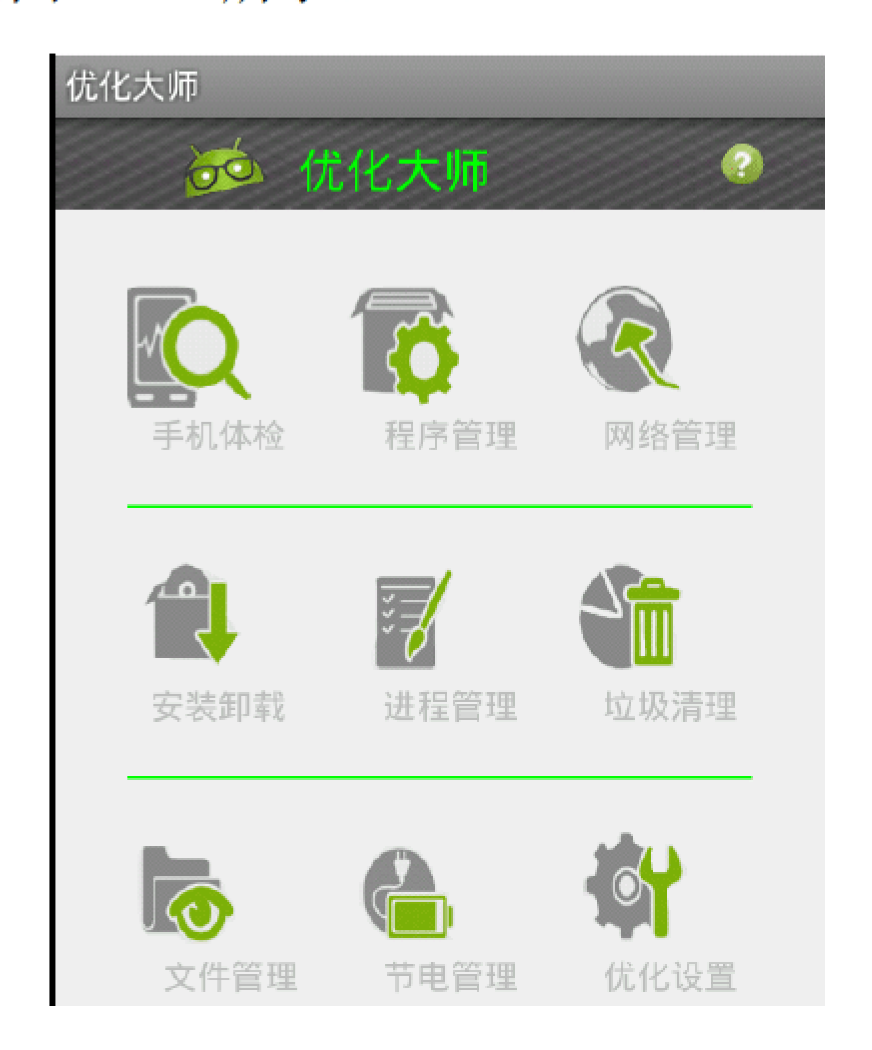


图 10-11 主界面执行效果

单击图 10-11 中的【进程管理】图标来到进程管理界面，如图 10-12 所示。



图 10-12 进程管理界面

单击图 10-12 中的某个进程后会弹出一个提示框，如图 10-13 所示。



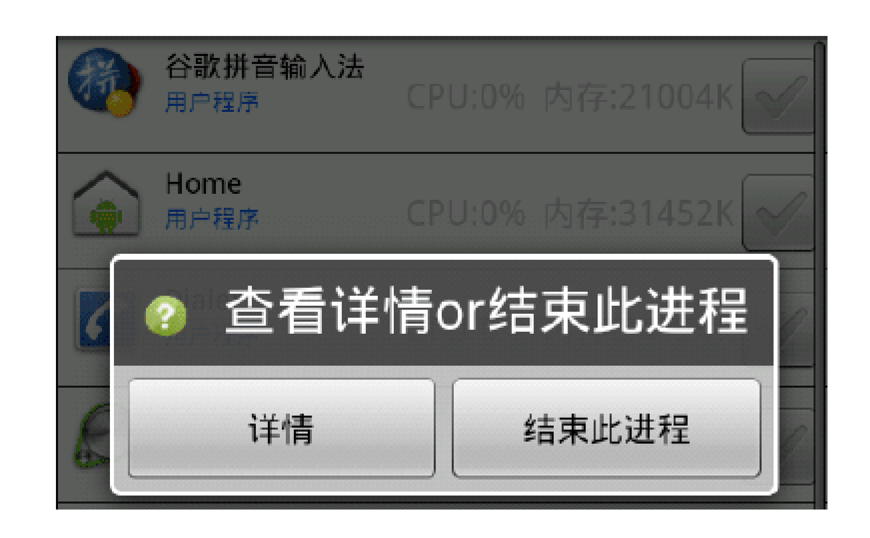


图 10-13　提示框

单击【详情】按钮可以在新界面中显示此进程的详细信息，如图 10-14 所示。单击【结束此进程】按钮后可以关闭进程。



图 10-14　某进程的详情

单击图 10-12 中顶部的【运行中服务】选项卡可以来到一个新界面，如图 10-15 所示。

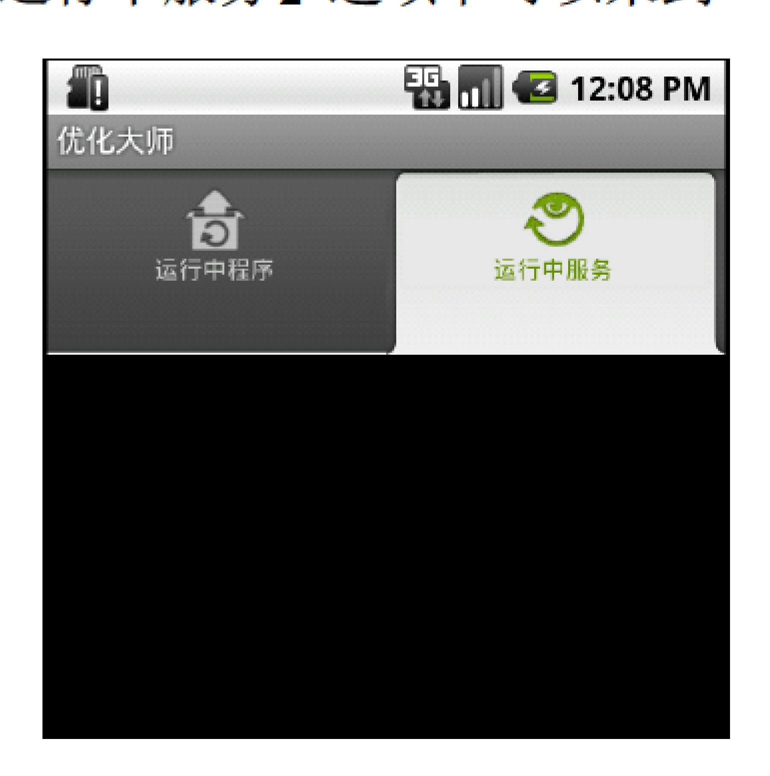


图 10-15　【运行中服务】界面

单击图 10-11 中的【文件管理】图标来到文件管理界面，如图 10-16 所示。



图 10-16　文件管理界面

单击图 10-16 顶部的【树图管理】标签后来到【树图管理】界面，如图 10-17 所示。

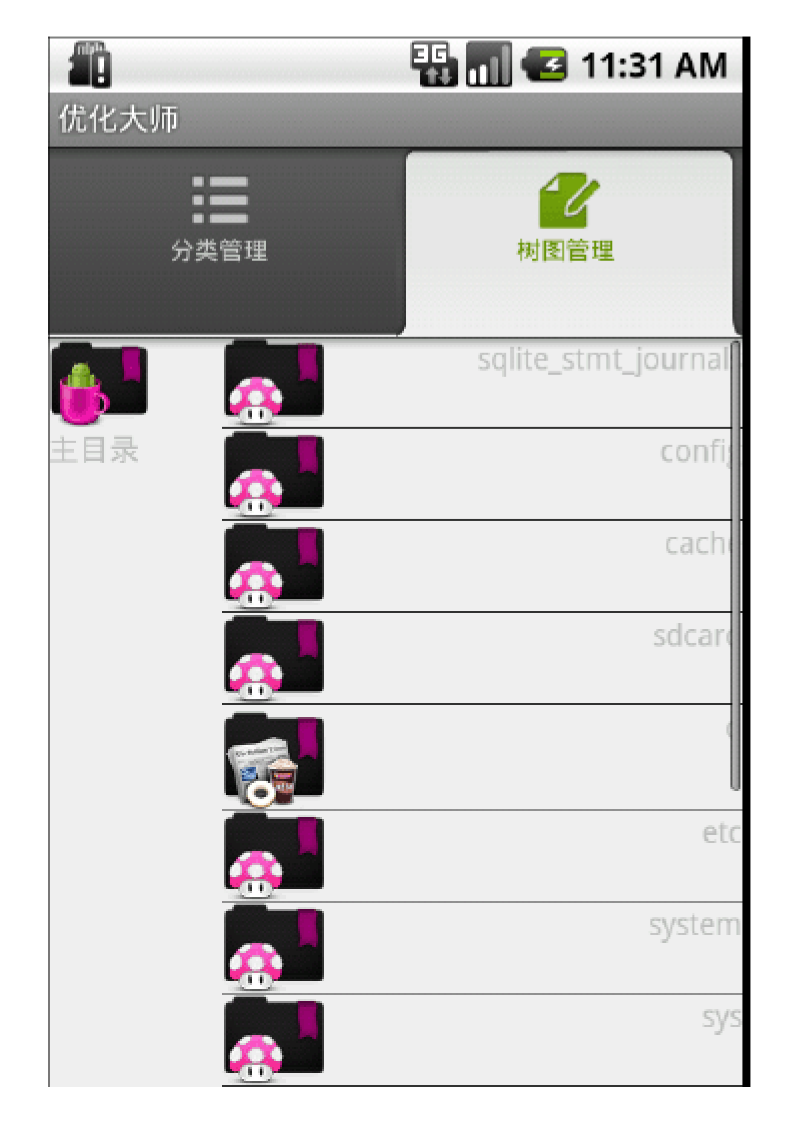


图 10-17 【树图管理】界面

在【树图管理】界面中可以查看某个文件夹下的所有子文件，例如图 10-18 显示了 system/app 目录下的文件。

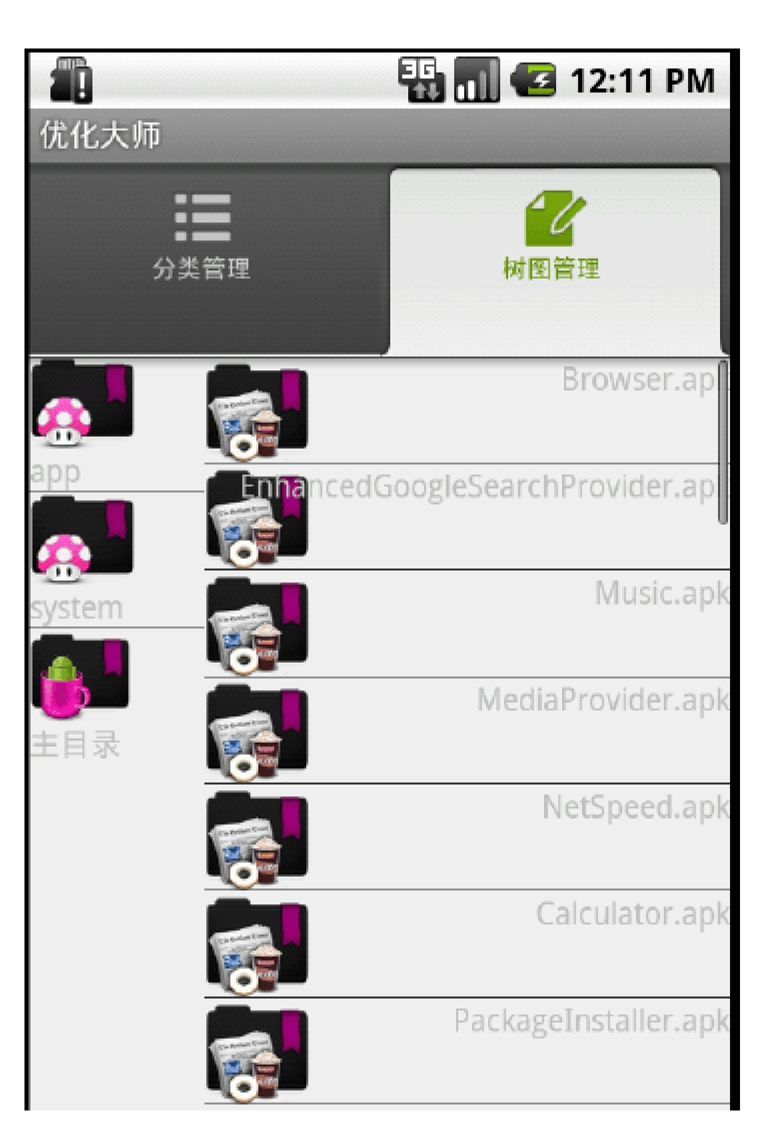


图 10-18 system/app 目录的子文件

Android

第11章

综合实例——手机地图系统

无论是出门旅行还是驴行，GPS 地图都十分有用，同时也给开发者带来了无限商机。Android 提供的地图(Map)功能可能是广大开发者最为关心的部分之一。在本章的内容中，将通过一个综合实例的实现过程来讲解地图的具体实现流程。本章源码保存在网络资源：daima\11\文件夹中。

11.1 项目分析

本项目实例的功能是，为用户提供需要的目标定位处理，即用户设置一个目标后，可以在后台启动一个 Service，能够定时读取 GPS 数据以获得用户目前所在的位置信息，并将其保存在数据库中。用户也可以选择其他目标信息，也能够将这些轨迹显示在 Map 地图上面。本项目的实现流程如图 11-1 所示。

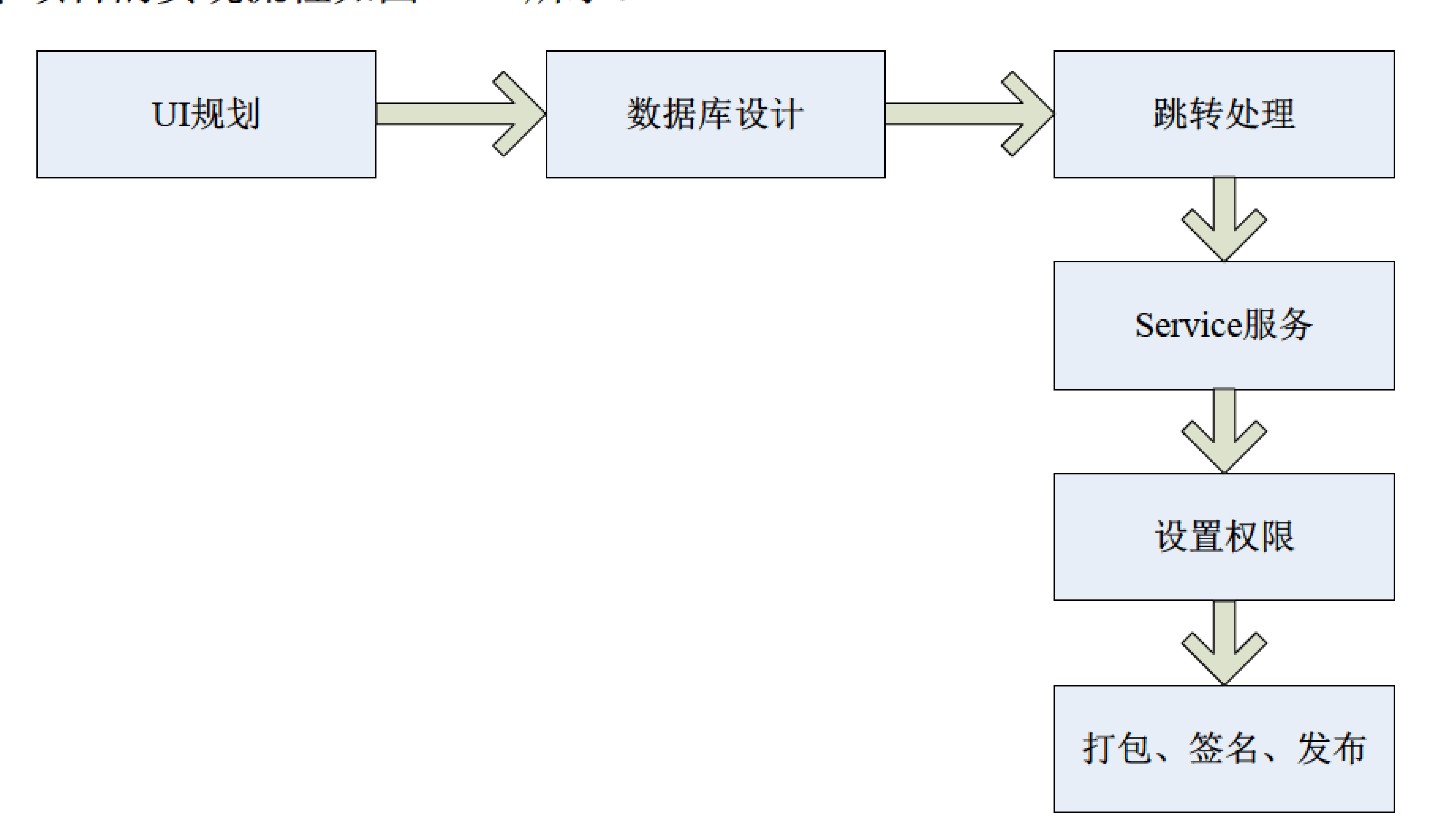


图 11-1　实现流程

11.1.1　规划 UI 界面

本项目 UI 界面的结构如图 11-2 所示。

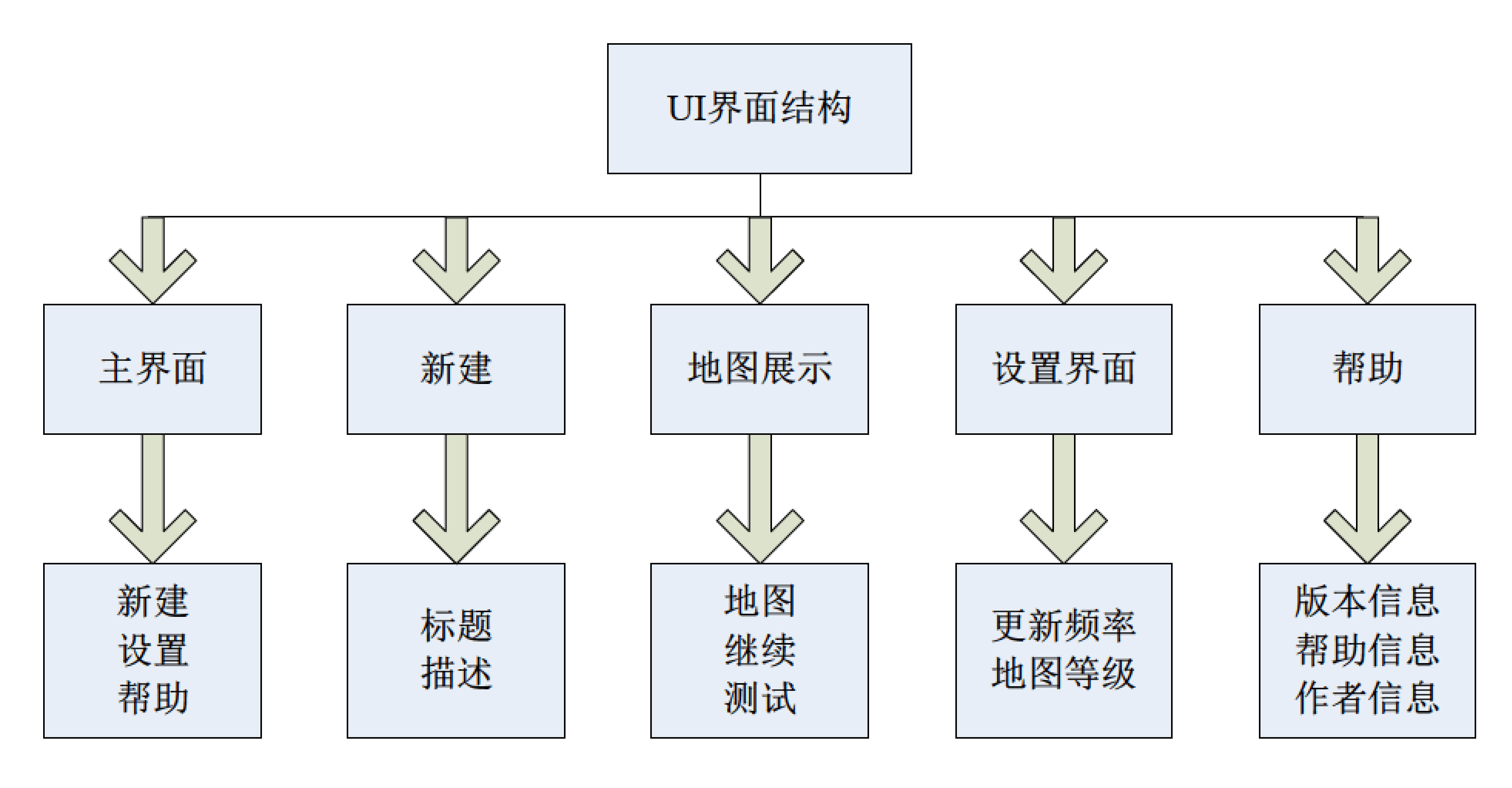


图 11-2　UI 界面结构

11.1.2　数据存储设计和优化

数据存储既可以通过文件系统实现，也可以通过专用数据库工具实现。但是为了保证系统的日后维护工作，本项目使用数据库工具方式。本项目使用的是最常见的 SQLite。

根据前面介绍的系统需求分析，本系统用到了三类数据，一种是目标名，一种是每次追踪时的位置信息，还有一种是配置信息。为此，在本系统需要设计两个表来存储数据，具体说明如下。

表 Tracks 用于存储目标信息，具体结构如表 11-1 所示。

表 11-1　表 Tracks 结构

属　性	类　型	说　明
id	INTEGER	主键
name	TEXT	名
desc	TEXT	说明
distance	LONG	距离
tracked_time	LONG	时间
locates_count	INTEGER	点数
created_at	INTEGER	创建时间
update_at	INTEGER	更新时间
avg_speed	LONG	平均速度
max_speed	LONG	最大速度

表 Locats 用于存储目标的位置信息，具体结构如表 11-2 所示。

表 11-2　表 Locats 结构

属　性	类　型	说　明
id	INTEGER	主键
track_id	INTEGER	跟踪的目标 ID
longitude	TEXT	维度
latiude	TEXT	经度
altitude	TEXT	偏差
created_at	INTEGER	创建时间

在编写处理 SQLite 中存储的数据时，实现优化工作需要做到如下三点。

(1) 对于单个表的单个列而言，如果都有形如 T.C=expr 这样的子句，并且都是用 OR 操作符连接起来，例如：

```
x = expr1 OR expr2 = x OR x = expr3
```

此时由于对于 OR，在 SQLite 中不能利用索引来优化，所以可以将它转换成如下带有 IN 操作符的子句：

```
x IN(expr1,expr2,expr3)
```

这样就可以用索引进行优化，效果很明显，但是如果在都没有索引的情况下 OR 语句执行效率会稍优于 IN 语句的效率。

(2) 如果一个子句的操作符是 BETWEEN，在 SQLite 中同样不能用索引进行优化，所以也要进行相应的等价转换，例如：

```
a BETWEEN b AND c
```

可以转换成：

```
(a BETWEEN b AND c) AND (a>=b) AND (a<=c)
```

在上述子句中，(a>=b) AND (a<=c)将被设为 dynamic，并且是(a BETWEEN b AND c)的子句，那么如果 BETWEEN 语句已经编码，那么子句就忽略不计，如果存在可利用的 index 使得子句已经满足条件，那么父句则被忽略。

(3) 如果一个单元的操作符是 LIKE，那么将做下面的转换：

将 x LIKE 'abc%'转换成：x>='abc'AND x<'abd'。因为在 SQLite 中的 LIKE 是不能用索引进行优化的，所以如果存在索引的话，则转换后和不转换相差很远，因为对 LIKE 不起作用，但如果不存在索引，那么 LIKE 在效率方面也还是比不上转换后的效率的。

11.2 具体实现

到此为止，一个项目的准备工作就做好了。接下来将开始介绍本项目的具体实现过程，希望读者认真体会每一段代码的功能和编写原理，为提高自己的开发水平做好准备。

11.2.1 新建工程

打开 Eclipse，依次选择 File | New | Android Project 命令，新建一个名为“map”的工程文件，如图 11-3 所示。

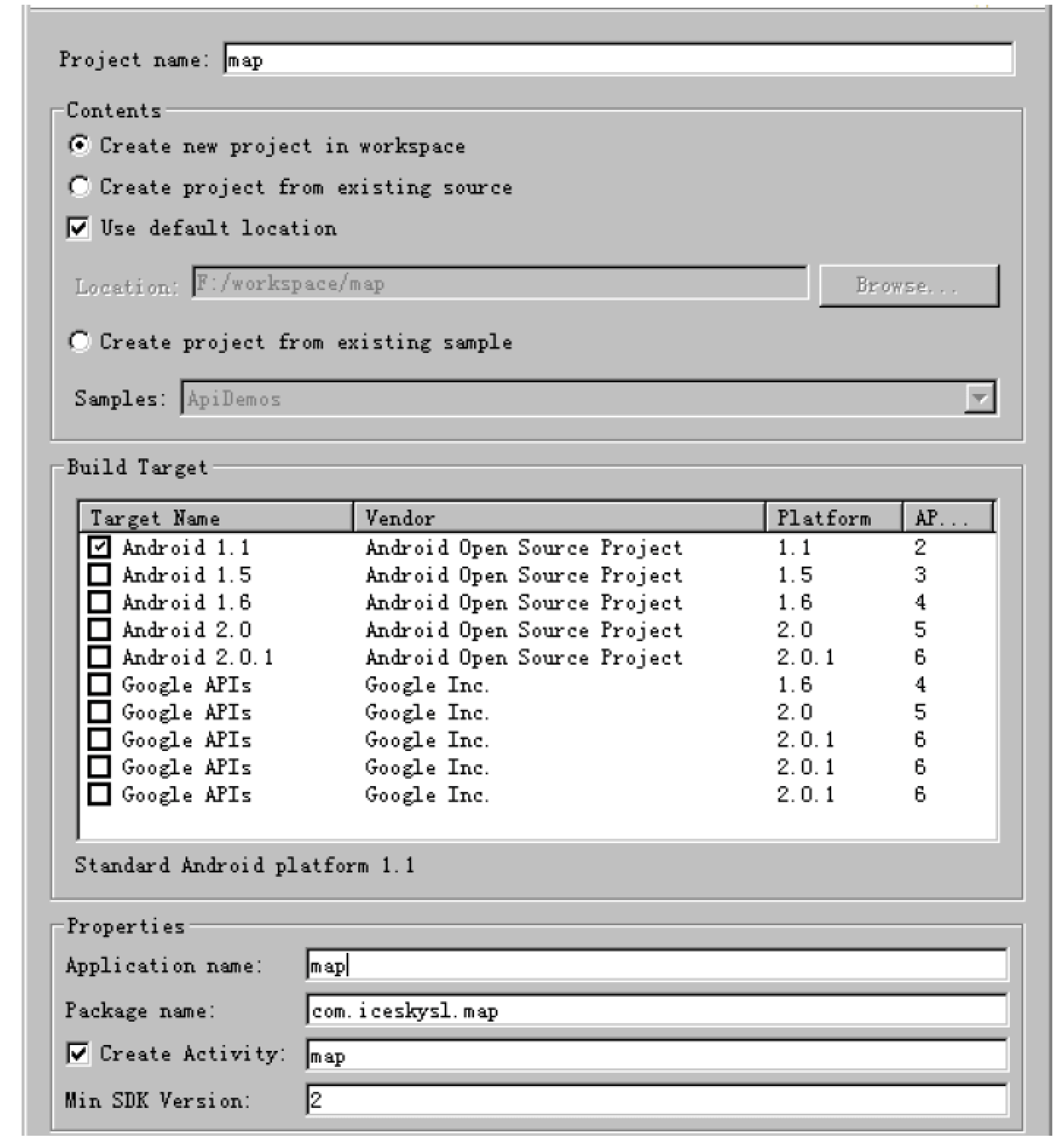


图 11-3　新建工程

11.2.2　主界面

主界面即项目执行后首先显示的界面，实现本项目主界面的流程如下。

(1) 编写主布局文件 main.xml，具体代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout width="fill parent"
   android:layout height="fill parent"
   >
<TextView
   android:layout width="fill parent"
   android:layout height="wrap content"
   android:text="@string/title"
   />

    <ListView android:id="@id/android:list"
       android:layout width="fill parent"
        android:layout height="wrap content"
        android:drawSelectorOnTop="false" />

    <TextView android:id="@+id/android:empty"
```

```
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:text="@string/start" />
</LinearLayout>
```

在 hierarchyviewer.bat 中的 UI 视图效果如图 11-4 所示。

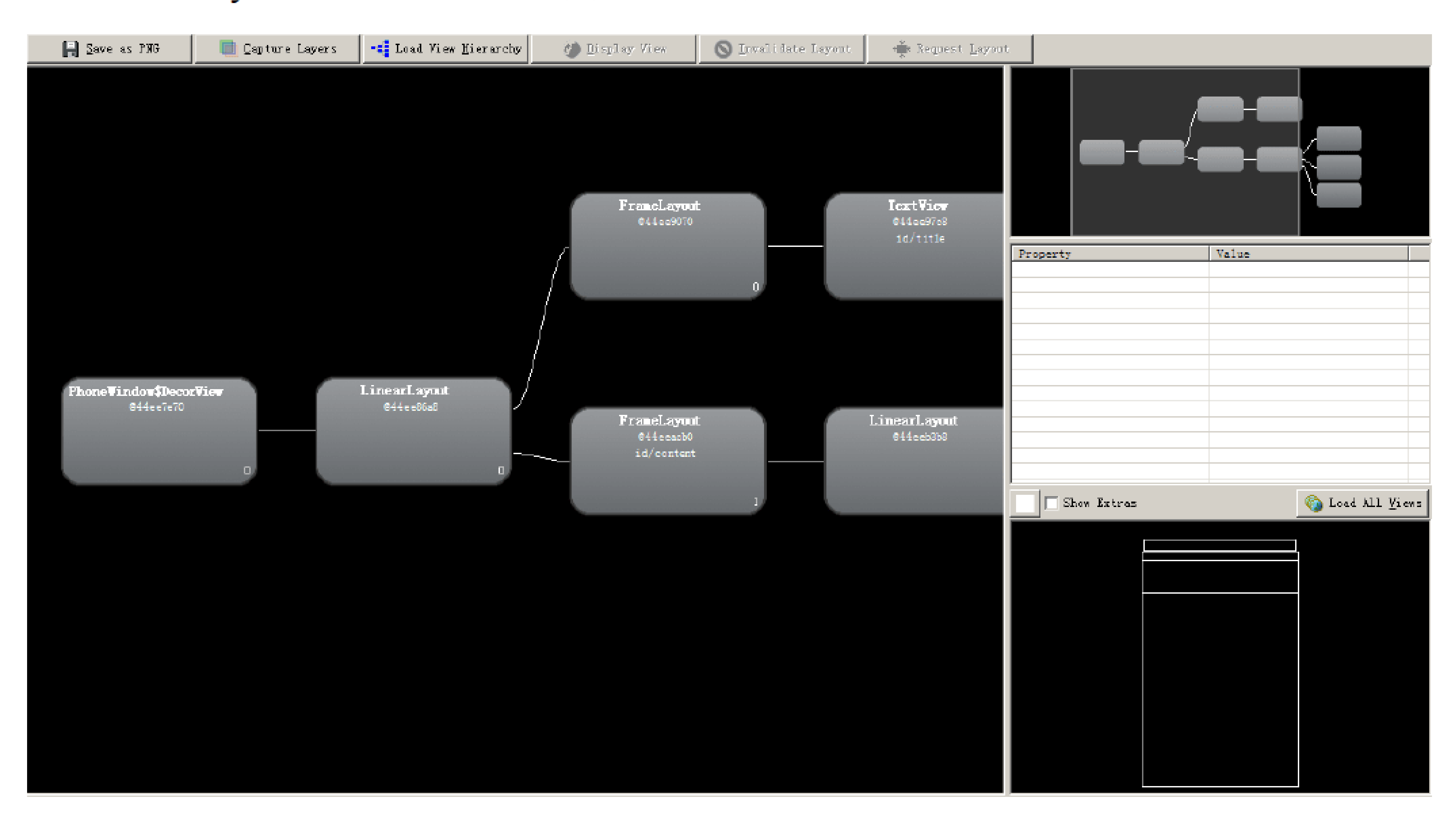


图 11-4　hierarchyviewer.bat 中的 UI 视图

(2) 编写一个“历史记录”的列表信息，只显示系统数据库内的前 10 条数据。其功能是文件 string.xml 实现的，具体代码如下。

```
    <string name="title">历史记录:</string>
    <string name="app name">111</string>
    <string name="app title">222</string>
    <string name="menu main">主页</string>
    <string name="menu new">新建</string>
    <string name="menu_con">继续</string>
    <string name="menu_del">删除</string>
    <string name="menu_setting">设置</string>
    <string name="menu_helps">帮助</string>
    <string name="menu_back">返回</string>
    <string name="menu_exit">退出</string>
```

(3) 编写 onCreate 方法，将以往的历史记录从数据库中读取出来，显示在列表中，然后使用 render_tracks()方法将数据库的历史记录读取出来，并更新到列表中去。具体代码如下。

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
         render tracks();
    }
```

(4) 在文件 iTracks.java 中编写实现菜单的代码，创建菜单框架和菜单被选中后的响应



方法。主要代码如下。

```
//定义菜单需要的常量
private static final int MENU_NEW = Menu.FIRST + 1;
private static final int MENU_CON = MENU_NEW + 1;
private static final int MENU_SETTING = MENU_CON + 1;
private static final int MENU_HELPS = MENU_SETTING + 1;
private static final int MENU_EXIT = MENU_HELPS + 1;
// 初始化菜单
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    Log.d(TAG, "onCreateOptionsMenu");

    super.onCreateOptionsMenu(menu);

    menu.add(0, MENU_NEW, 0, R.string.menu_new).setIcon(
            R.drawable.new_track).setAlphabeticShortcut('N');
    menu.add(0, MENU_CON, 0, R.string.menu_con).setIcon(
            R.drawable.con_track).setAlphabeticShortcut('C');
    menu.add(0, MENU_SETTING, 0, R.string.menu_setting).setIcon(
            R.drawable.setting).setAlphabeticShortcut('S');
    menu.add(0, MENU_HELPS, 0, R.string.menu_helps).setIcon(
            R.drawable.helps).setAlphabeticShortcut('H');
    menu.add(0, MENU_EXIT, 0, R.string.menu_exit).setIcon(
            R.drawable.exit).setAlphabeticShortcut('E');
    return true;
}
// 当一个菜单被选中的时候调用
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    Intent intent = new Intent();
    switch (item.getItemId()) {
    case MENU_NEW:
        intent.setClass(iTracks.this, NewTrack.class);
        startActivity(intent);
        return true;
    case MENU_CON:
        //TODO: 继续跟踪选择的记录
        conTrackService();
        return true;
    case MENU_SETTING:
        intent.setClass(iTracks.this, Setting.class);
        startActivity(intent);
        return true;
    case MENU_HELPS:
        intent.setClass(iTracks.this, Helps.class);
        startActivity(intent);
        return true;
    case MENU_EXIT:
        finish();
        break;
```

```
        }
        return true;
    }
```

至此主界面的设计工作全部结束，执行后的效果如图 11-5 所示。



图 11-5　主界面

11.2.3　新建界面

当在图 11-5 中单击【新建】按钮后进入新建目标记录界面，此模块的实现流程如下。

(1) 编写布局文件 new_track.xml，分别用 TextView 来显示提示信息，用 EditText 来接收用户的输入。主要代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout width="fill parent"
    android:layout height="fill parent">
    <TextView android:layout width="fill parent"
        android:layout height="wrap content"
        android:text="@string/new tips" />
    <TextView android:layout width="fill parent"
        android:layout_height="wrap_content"
        android:text="@string/new name" />
    <EditText android:id="@+id/new name"
    android:layout width="fill parent"
        android:layout height="wrap content"
        android:text="" />
    <TextView android:layout width="fill parent"
        android:layout_height="wrap_content"
```

```
        android:text="@string/new desc" />
    <EditText android:id="@+id/new desc"
        android:layout width="fill parent"
        android:layout height="wrap content"
        android:layout weight="1"
        android:scrollbars="vertical"/>
    <Button android:id="@+id/new submit"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:text="@string/new submit" />
</LinearLayout>
```

(2) 编写处理文件 NewTrack.java，首先在方法 onCreate 中设置了其关联的 layout；然后调用 findViewsByid()来获取名字和 EditText 组件，并获取提交按钮；最后，定义了一个 Button.OnClickListener new_track 对象，实现其 onClick 方法。文件 NewTrack.java 的主要代码如下。

```
public class NewTrack extends Activity {
   private static final String TAG = "NewTrack";
    private Button button new;
    private EditText field new name;
    private EditText field new desc;

    private TrackDbAdapter mDbHelper;

    /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.new track);
    setTitle(R.string.menu new);
        findViews();
        setListensers();

        mDbHelper = new TrackDbAdapter(this);
        mDbHelper.open();
   }

    @Override
    protected void onStop(){
        super.onStop();
        Log.d(TAG, "onStop");
        mDbHelper.close();
    }

    private void findViews() {
        Log.d(TAG, "find Views");
        field new name = (EditText) findViewById(R.id.new name);
        field_new_desc = (EditText) findViewById(R.id.new_desc);
        button new = (Button) findViewById(R.id.new submit);
    }
```

```
    // Listen for button clicks
    private void setListensers() {
        Log.d(TAG, "set Listensers");
        button new.setOnClickListener(new track);
    }

    private Button.OnClickListener new track = new Button.OnClickListener() {
        public void onClick(View v) {
            Log.d(TAG, "onClick new track..");
            try {
                String name = (field new name.getText().toString());
                String desc = (field new desc.getText()
                        .toString());
                if (name.equals("")) {
                    Toast.makeText(NewTrack.this,
                            getString(R.string.new name null),
                            Toast.LENGTH SHORT).show();
                } else {
                    // TODO 调用存储接口保存到数据库并启动 service
                    Long row id = mDbHelper.createTrack(name, desc);
                    Log.d(TAG, "row id="+row id);

                    Intent intent = new Intent();
                    intent.setClass(NewTrack.this, ShowTrack.class);
                    intent.putExtra(TrackDbAdapter.KEY ROWID, row id);
                    intent.putExtra(TrackDbAdapter.NAME, name);
                    intent.putExtra(TrackDbAdapter.DESC, desc);

                    startActivity(intent);
                }
            } catch (Exception err) {
                Log.e(TAG, "error: " + err.toString());
                Toast.makeText(NewTrack.this, getString(R.string.new fail),
                        Toast.LENGTH SHORT).show();
            }
        }
    };
}
```

至此，本模块功能的设计工作介绍完毕，执行后的效果如图 11-6 所示。



图 11-6　新建界面

11.2.4　设置界面

当在图 11-5 中单击【设置】按钮后弹出系统设置界面，此模块的实现流程如下。

(1) 编写布局文件 setting.xml，通过 Spinner 组件实现了一个供用户使用的下拉菜单。主要代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout width="fill parent"
    android:layout height="fill parent">
    <TextView android:id="@+id/setting tips"
        android:layout width="fill parent"
         android:layout height="wrap content"
         android:text="" />
    <TextView android:layout width="fill parent"
         android:layout height="wrap content"
         android:text="@string/setting gps" />
            <Spinner android:id="@+id/setting_gps"
        android:layout width="fill parent"
        android:layout height="wrap content"
        android:drawSelectorOnTop="true"
        android:prompt="@string/spinner gps prompt"
    />
    <TextView android:layout width="fill parent"
         android:layout height="wrap content"
         android:text="@string/setting map level" />
    <Spinner android:id="@+id/setting map level"
        android:layout width="fill parent"
        android:layout height="wrap content"
        android:drawSelectorOnTop="true"
        android:prompt="@string/spinner map prompt"
    />
    <Button android:id="@+id/setting submit"
         android:layout width="wrap content"
         android:layout height="wrap content"
         android:text="@string/setting submit" />
</LinearLayout>
```

(2) 编写处理文件 Setting.java，此文件的实现流程如下。

① 声明需要的变量，在 onCreat 中绑定 setting.xml 为其布局模板。

② 使用 setContentView()设定其对应的布局文件 setting.xml，使用 setTitle()设定其标题，进一步调用 findViews()查询到需要的操作组件，并调用 setListensers()给按钮设定单击监听器，最后调用 restorePrefs()将默认值或用户的历史选择值显示出来。

③ 使用 findViews()找到需要用到的组件。

文件 Setting.java 的主要代码如下。

```
public class Setting extends Activity {
   private static final String TAG = "Setting";
```

```
//定义菜单需要的常量
private static final int MENU MAIN = Menu.FIRST + 1;
private static final int MENU NEW = MENU MAIN + 1;
private static final int MENU BACK = MENU NEW + 1;;

// 保存个性化设置
public static final String SETTING INFOS = "SETTING Infos";
public static final String SETTING GPS = "SETTING Gps";
public static final String SETTING MAP = "SETTING Map";
public static final String SETTING GPS POSITON = "SETTING Gps p";
public static final String SETTING MAP POSITON = "SETTING Map p";

private Button button setting submit;
private Spinner field setting gps;
private Spinner field setting map level;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.setting);
setTitle(R.string.menu setting);
     findViews();
     setListensers();
     restorePrefs();
}

private void findViews() {
     Log.d(TAG, "find Views");
    button setting submit = (Button) findViewById(R.id.setting submit);
     field setting gps = (Spinner) findViewById(R.id.setting gps);
     ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(
           this, R.array.gps, android.R.layout.simple spinner item);
     adapter.setDropDownViewResource
        (android.R.layout.simple spinner dropdown item);
     field setting gps.setAdapter(adapter);

     field setting map level = (Spinner) findViewById(R.id.setting map level);
     ArrayAdapter<CharSequence> adapter2 =
         ArrayAdapter.createFromResource(
           this, R.array.map, android.R.layout.simple spinner item);
     adapter2.setDropDownViewResource
         (android.R.layout.simple spinner dropdown item);
    field setting map level.setAdapter(adapter2);
}

private void setListensers() {
     Log.d(TAG, "set Listensers");
     button setting submit.setOnClickListener(setting submit);
}
private Button.OnClickListener setting submit = new Button.OnClickListener() {
     public void onClick(View v) {
```



```
            Log.d(TAG, "onClick new track..");
            try {
               String gps = (field setting gps.getSelectedItem().toString());
                String map = (field setting map level.getSelectedItem()
                        .toString());
                if (gps.equals("") || map.equals("")) {
                    Toast.makeText(Setting.this,
                            getString(R.string.setting null),
                            Toast.LENGTH SHORT).show();
                } else {
                    //保存设定
                    storePrefs();
                    Toast.makeText(Setting.this,
                            getString(R.string.setting ok),
                            Toast.LENGTH SHORT).show();
                    //跳转到主界面
                    Intent intent = new Intent();
                    intent.setClass(Setting.this, iTracks.class);
                    startActivity(intent);
                }
            } catch (Exception err) {
                Log.e(TAG, "error: " + err.toString());
               Toast.makeText(Setting.this, getString(R.string.setting fail),
                        Toast.LENGTH SHORT).show();
            }
        }
    };

    private void restorePrefs() {
       SharedPreferences settings = getSharedPreferences(SETTING INFOS, 0);
        int setting gps p = settings.getInt(SETTING GPS POSITON, 0);
        int setting map level p = settings.getInt(SETTING MAP POSITON, 0);
        Log.d(TAG, "restorePrefs: setting gps= "+ setting gps p + ",
            setting map level=" + setting map level p);

        if (setting gps p != 0 && setting map level p != 0) {
            field setting gps.setSelection(setting gps p);
            field setting map level.setSelection(setting map level p);
            button setting submit.requestFocus();
        }else if(setting gps p != 0 ){
            field setting gps.setSelection(setting gps p);
            field setting map level.requestFocus();
        }else if(setting map level p != 0){
            field setting map level.setSelection(setting map level p);
            field setting gps.requestFocus();
        }else{
            field setting gps.requestFocus();
        }
    }

    protected void onStop(){
```

```
        super.onStop();
        Log.d(TAG, "save setting infos");
        storePrefs();
    }

    //保存个人设置
    private void storePrefs() {
        Log.d(TAG, "storePrefs setting infos");
       SharedPreferences settings = getSharedPreferences(SETTING INFOS, 0);
        settings.edit()
            .putString(SETTING GPS,
                field setting gps.getSelectedItem().toString())
            .putString(SETTING MAP,
                field setting map level.getSelectedItem().toString())
            .putInt(SETTING GPS POSITON,
                field setting gps.getSelectedItemPosition())
            .putInt(SETTING MAP POSITON,
                field setting map level.getSelectedItemPosition())
            .commit();
    }

    // 初始化菜单
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        menu.add(0, MENU MAIN, 0, R.string.menu main).setIcon(
                R.drawable.icon).setAlphabeticShortcut('M');
        menu.add(0, MENU NEW, 0, R.string.menu new).setIcon(
                R.drawable.new track).setAlphabeticShortcut('N');
        menu.add(0, MENU BACK, 0, R.string.menu back).setIcon(
                R.drawable.back).setAlphabeticShortcut('E');
        return true;
    }

    // 当一个菜单被选中的时候调用
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        Intent intent = new Intent();
        switch (item.getItemId()) {
        case MENU NEW:
            intent.setClass(Setting.this, NewTrack.class);
            startActivity(intent);
            return true;
        case MENU MAIN:
            intent.setClass(Setting.this, iTracks.class);
            startActivity(intent);
            return true;
        case MENU BACK:
           finish();
            break;
        }
```

```
        return true;
    }

}
```

此处下拉框中的内容是预先设置好的，这些内容在文件 array.xml 中定义，对应代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Used in View/setting.java -->
    <string-array name="gps">
        <item>1</item>
        <item>10</item>
        <item>20</item>
        <item>30</item>
        <item>40</item>
        <item>50</item>
    </string-array>
    <string-array name="map">
        <item>2</item>
        <item>3</item>
        <item>4</item>
        <item>5</item>
        <item>20</item>
        <item>30</item>
        <item>41</item>
        <item>52</item>
        <item>63</item>
        <item>74</item>
        <item>85</item>
        <item>96</item>
    </string-array>
</resources>
```

至此，本模块的设计工作介绍完毕，执行后的效果如图 11-7 所示。



图 11-7　设置界面

11.2.5 帮助界面

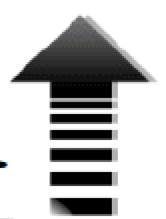
当在图 11-5 中单击【帮助】按钮后弹出系统默认的帮助界面，此模块的实现流程如下。

(1) 编写布局文件 helps.xml，通过 TextView 显示了各条帮助信息。主要代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/version"
    />
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/version_text"
    />
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/helps_infos"
    />
   <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:autoLink="all"
    android:text="@string/helps_text"
 />
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/author"
    />
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:autoLink="all"
    android:text="@string/author_text"
 />
</LinearLayout>
```

(2) 编写处理文件 helps.java，在此首先在 onCreate 方法中设定其对应的布局文件为 helps.xml，然后添加菜单和菜单对应的功能。主要代码如下。

```
public class Helps extends Activity {
    //定义菜单需要的常量
```



```
    private static final int MENU MAIN = Menu.FIRST + 1;
    private static final int MENU NEW = MENU MAIN + 1;
    private static final int MENU BACK = MENU NEW + 1;;
   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.helps);
    setTitle(R.string.menu helps);
   }
    // 初始化菜单
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        menu.add(0, MENU MAIN, 0, R.string.menu main).setIcon(
                R.drawable.icon).setAlphabeticShortcut('M');
        menu.add(0, MENU NEW, 0, R.string.menu new).setIcon(
                R.drawable.new track).setAlphabeticShortcut('N');
        menu.add(0, MENU BACK, 0, R.string.menu back).setIcon(
                R.drawable.back).setAlphabeticShortcut('E');
        return true;
    }

    // 当一个菜单被选中的时候调用
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        Intent intent = new Intent();
        switch (item.getItemId()) {
        case MENU NEW:
            intent.setClass(Helps.this, NewTrack.class);
            startActivity(intent);
            return true;
        case MENU MAIN:
            intent.setClass(Helps.this, iTracks.class);
            startActivity(intent);
            return true;
        case MENU BACK:
            finish();
            break;
        }
        return true;
    }
}
```

至此，本模块的设计工作介绍完毕，执行后的效果如图 11-8 所示。

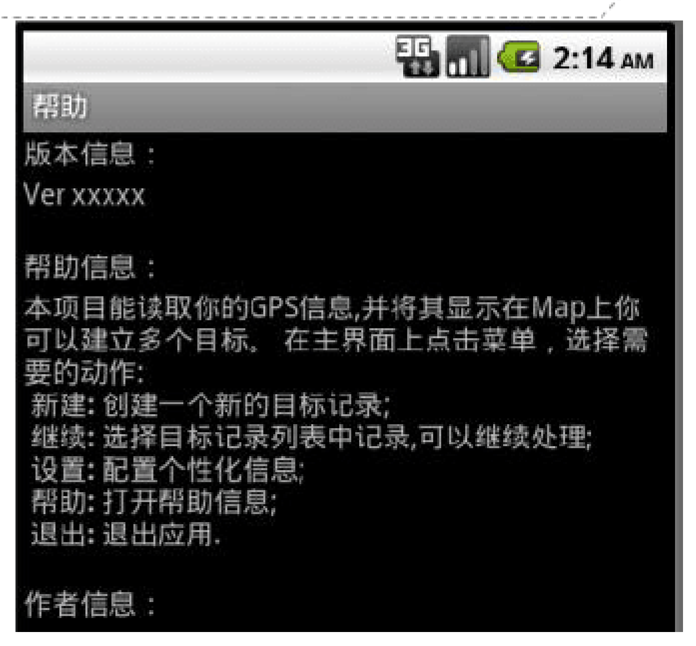


图 11-8　帮助界面

11.2.6　地图界面

前面介绍的都是主菜单中的选项，接下来开始介绍本实例的核心功能——在 Android 手机中显示 Google 地图。在实现之前需要先进行相关的设置操作。

1. 申请 APIKey

(1) 打开 Eclipse，依次选择 Windows | Preferences 命令，在打开的对话框中选择 Android 选项下的 Build 项，查看默认的 debug keystore 位置，如图 11-9 所示。

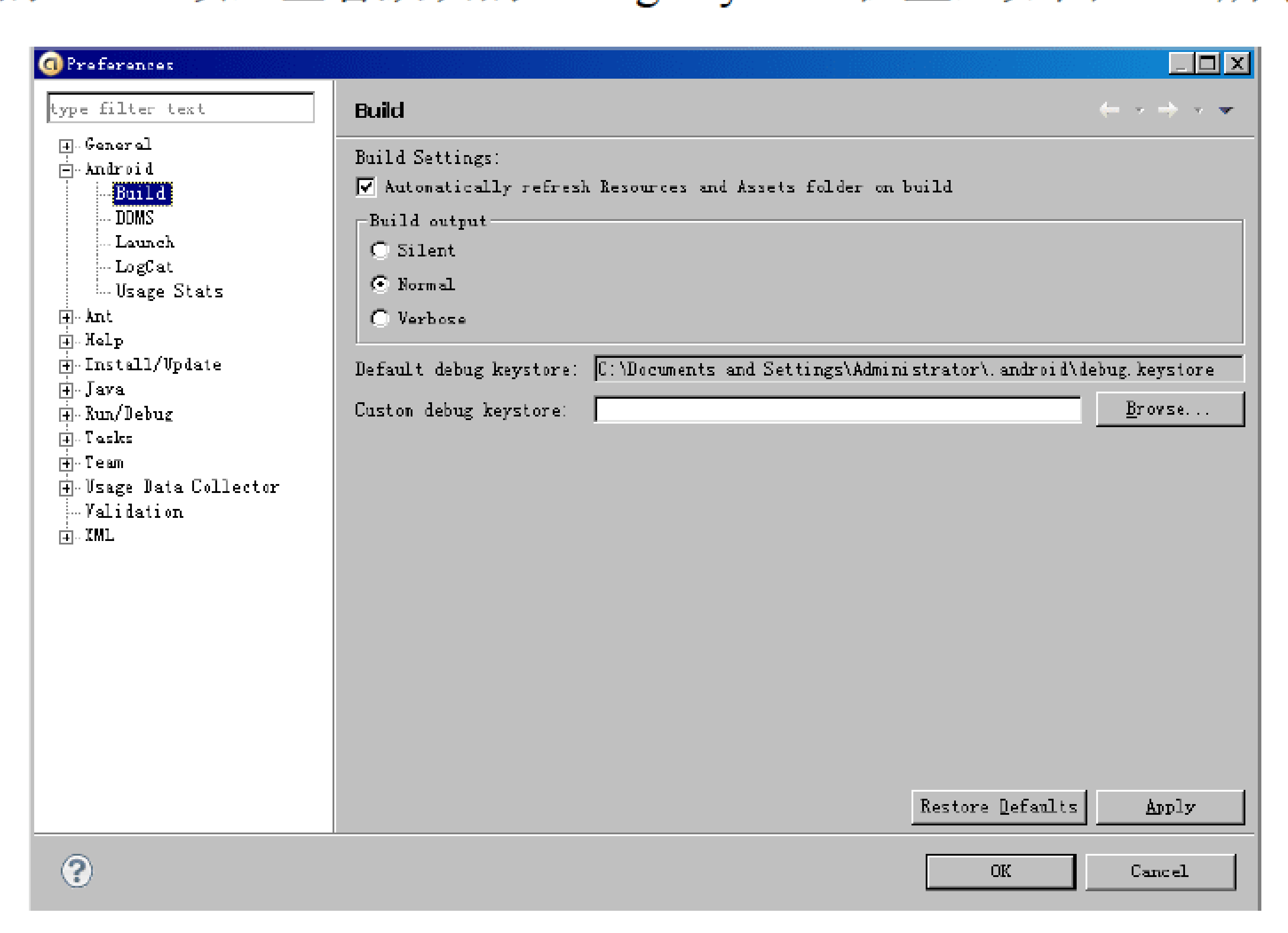


图 11-9　debug keystore 位置

(2) 打开 cmd，在 cmd 中执行：

```
keytool -list -alias androiddebugkey -keystore "C:\Documents and
Settings\Administrator\.android\debug.keystore" -storepass android -
keypass android
```

通过上述命令获取 MD5 指纹，此时系统会提示输入 keystore 代码，输入“android”后会显示要获取的指纹。

(3) 打开网址，输入得到的 MD5 指纹，然后单击 Generate API Key 按钮申请获取 API Key，如图 11-10 所示。

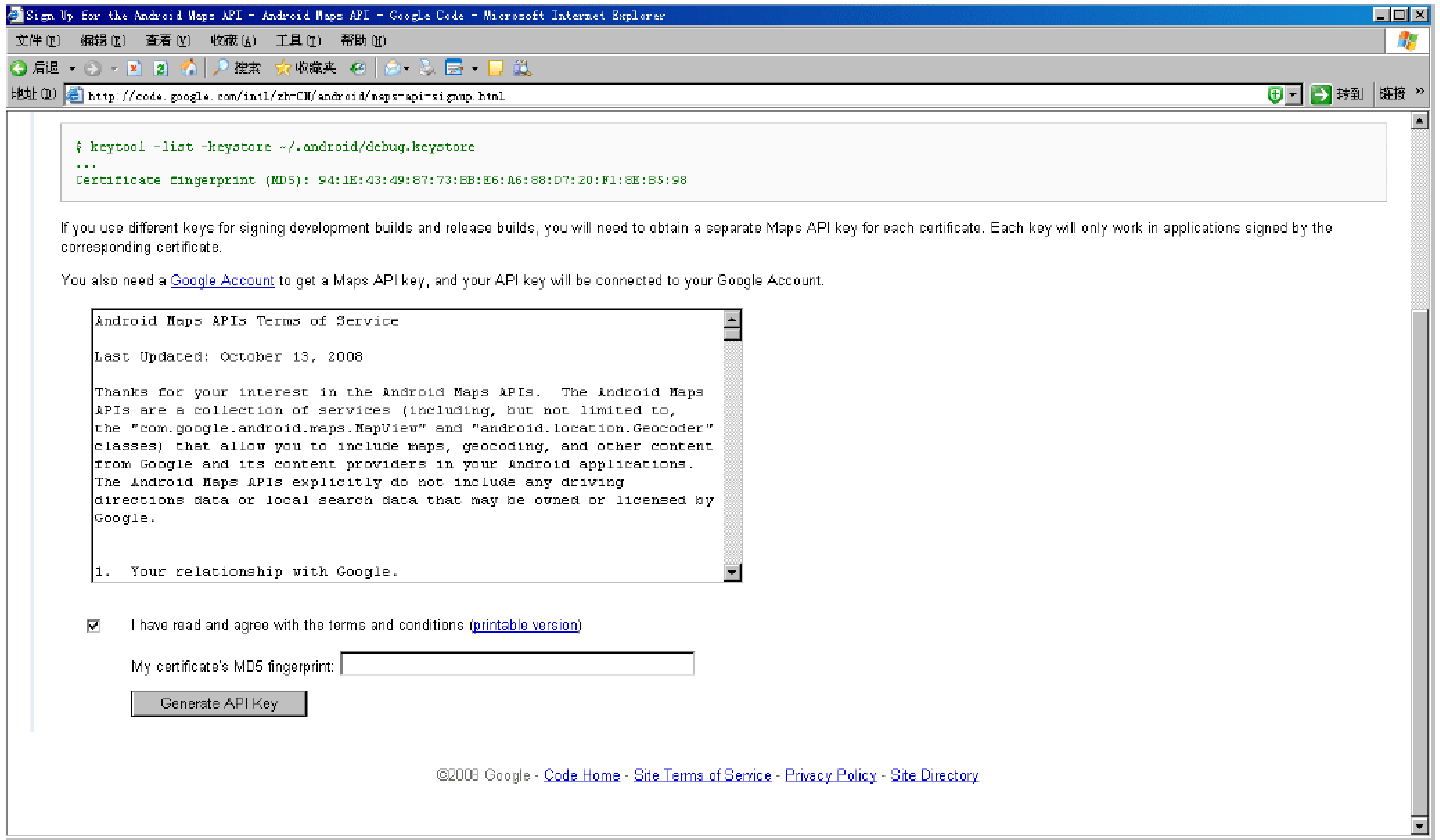


图 11-10 获取 APIKey

2. 具体编码

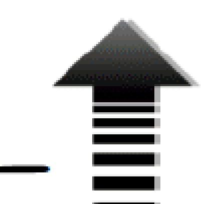
(1) 编写布局文件 show_track.xml，在此通过 MapView 组件来显示地图，并通过设置的按钮来控制地图，例如放大、缩小、移动和模式的转换。具体代码如下。

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android=
   "http://schemas.android.com/apk/res/android"
   android:layout width="fill parent"
   android:layout height="fill parent"
   >
<view android:id="@+id/mv"
   class="com.xxx.android.maps.MapView"
   android:layout width="fill parent"
   android:layout height="fill parent"
   android:layout weight="1"
   android:apiKey=" 0by7ffx8jX0C4kaxou6vckW6pkvss4ZwxjYIofg"
   />
<LinearLayout xmlns:android=
   "http://schemas.android.com/apk/res/android"
   android:orientation="horizontal"
   android:layout width="fill parent"
   android:layout height="wrap content"
   android:background="#550000ff"
   android:padding="1px"
   >
<Button android:id="@+id/sat"
   android:layout width="wrap content"
   android:layout_height="wrap_content"
   android:layout_marginLeft="40px"
```

```
    android:text="@string/satellite" />
<Button android:id="@+id/traffic"
     android:layout width="wrap content"
     android:layout height="wrap content"
     android:text="@string/traffic" />
<Button android:id="@+id/streetview"
     android:layout width="wrap content"
     android:layout height="wrap content"
     android:text="@string/street" />

<Button android:id="@+id/gps"
     android:layout width="wrap content"
     android:layout height="wrap content"
     android:text="GPS" />
</LinearLayout>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout width="wrap content"
    android:layout height="fill parent"
    android:background="#550000ff"
    android:padding="1px"
    >
<Button android:id="@+id/zin"
    android:layout width="wrap content"
    android:layout height="wrap content"
    android:layout marginTop="30px"
    android:text="+" />
<Button android:id="@+id/zout"
     android:layout width="wrap content"
     android:layout height="wrap content"
     android:text="-" />
<Button android:id="@+id/pann"
     android:layout width="wrap content"
     android:layout height="wrap content"
     android:text="N" />
<Button android:id="@+id/pane"
     android:layout width="wrap content"
     android:layout height="wrap content"
     android:text="E" />

<Button android:id="@+id/panw"
     android:layout width="wrap content"
     android:layout height="wrap content"
     android:text="W" />
<Button android:id="@+id/pans"
     android:layout width="wrap content"
     android:layout height="wrap content"
     android:text="S" />
</LinearLayout>
</FrameLayout>
```



(2) 编写处理文件 ShowTrack.java，此文件的具体实现流程如下。

① 通过 findViews 来确定要使用的控件，并绑定需要响应的事件。

② 通过 findViews 实现对地图的处理，首先获取布局中的 MapView，使用 getController 得到一个 MapController，然后注册一个基于 locationListener 的 MyLocationListener。

③ 实现处理按钮的处理方法的代码，具体原理比较简单，即首先获取地图中心，然后向 4 个方向移动 1/4 距离。

④ 单击 GPS 按钮后响应方法 centerOnGPSPosition，能够将地图定位到当前 GPS 指定的位置。

⑤ 通过 Overylay 抽象类重载实现其 draw 绘制方法。

文件 ShowTrack.java 的主要代码如下。

```
public class ShowTrack extends MapActivity {
    // 定义菜单需要的常量
    private static final int MENU NEW = Menu.FIRST + 1;
    private static final int MENU CON = MENU NEW + 1;
    private static final int MENU DEL = MENU CON + 1;
    private static final int MENU MAIN = MENU DEL + 1;

    private TrackDbAdapter mDbHelper;
    private LocateDbAdapter mlcDbHelper;

    private static final String TAG = "ShowTrack";
    private static MapView mMapView;
    private MapController mc;

    protected MyLocationOverlay mOverlayController;
    private Button mZin;
    private Button mZout;
    private Button mPanN;
    private Button mPanE;
    private Button mPanW;
    private Button mPanS;
    private Button mGps;
    private Button mSat;
    private Button mTraffic;
    private Button mStreetview;
    private String mDefCaption = "";
    private GeoPoint mDefPoint;

    private LocationManager lm;
    private LocationListener locationListener;

    private int track id;
    private Long rowId;

    public void onCreate(Bundle icicle) {
```

```
        super.onCreate(icicle);
        setContentView(R.layout.show track);
        findViews();
        centerOnGPSPosition();
        revArgs();
        paintLocates();
        startTrackService();
    }

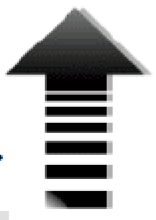
    private void startTrackService() {
        Intent i = new Intent("com.iceskysl.iTracks.START TRACK SERVICE");
        i.putExtra(LocateDbAdapter.TRACKID, track id);
       startService(i);
    }

    private void stopTrackService() {
       stopService(new Intent("com.iceskysl.iTracks.START TRACK SERVICE"));
    }

    private void paintLocates() {
        mlcDbHelper = new LocateDbAdapter(this);
        mlcDbHelper.open();
       Cursor mLocatesCursor = mlcDbHelper.getTrackAllLocates(track id);
        startManagingCursor(mLocatesCursor);
        Resources resources = getResources();
        Overlay overlays = new LocateOverLay(resources
                .getDrawable(R.drawable.icon), mLocatesCursor);
        mMapView.getOverlays().add(overlays);
        mlcDbHelper.close();
    }

    private void revArgs() {
        Log.d(TAG, "revArgs.");
        Bundle extras = getIntent().getExtras();
        if (extras != null) {
            String name = extras.getString(TrackDbAdapter.NAME);
            rowId = extras.getLong(TrackDbAdapter.KEY ROWID);
            track id = rowId.intValue();
            Log.d(TAG, "rowId=" + rowId);
            if (name != null) {
                setTitle(name);
            }
        }
    }
    protected boolean isRouteDisplayed() {
        return false;
    }

    private void findViews() {
        Log.d(TAG, "find Views");
        mMapView = (MapView) findViewById(R.id.mv);
```



```
        mc = mMapView.getController();

        SharedPreferences settings = getSharedPreferences
            (Setting.SETTING INFOS, 0);
       String setting gps = settings.getString(Setting.SETTING MAP, "10");
        mc.setZoom(Integer.parseInt(setting gps));
        mPanE = (Button) findViewById(R.id.sat);
        mPanE.setOnClickListener(new OnClickListener() {
            // @Override
            public void onClick(View arg0) {
                panEast();
            }
        });
        mZin = (Button) findViewById(R.id.zin);
        mZin.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
                zoomIn();
            }
        });
        mZout = (Button) findViewById(R.id.zout);
        mZout.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
                zoomOut();
            }
        });
        mPanN = (Button) findViewById(R.id.pann);
        mPanN.setOnClickListener(new OnClickListener() {
            // @Override
            public void onClick(View arg0) {
                panNorth();
            }
        });

        mPanE = (Button) findViewById(R.id.pane);
        mPanE.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
                panEast();
            }
        });

        // Set up the button for "Pan West"
        mPanW = (Button) findViewById(R.id.panw);
        mPanW.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
                panWest();
            }
        });

        mPanS = (Button) findViewById(R.id.pans);
        mPanS.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
```

```
                panSouth();
            }
        });
        mGps = (Button) findViewById(R.id.gps);
        mGps.setOnClickListener(new OnClickListener() {
            // @Override
            public void onClick(View arg0) {
                centerOnGPSPosition();
            }
        });
        mSat = (Button) findViewById(R.id.sat);
        mSat.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
                toggleSatellite();
            }
        });
        mTraffic = (Button) findViewById(R.id.traffic);
        mTraffic.setOnClickListener(new OnClickListener() {
            public void onClick(View arg0) {
                toggleTraffic();
            }
        });
        mStreetview = (Button) findViewById(R.id.streetview);
        mStreetview.setOnClickListener(new OnClickListener() {
            // @Override
            public void onClick(View arg0) {
                toggleStreetView();
            }
        });
       lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
        locationListener = new MyLocationListener();
        lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0,
                locationListener);
    }
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        Log.d(TAG, "onKeyDown");
        if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
            panWest();
            return true;
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
            panEast();
            return true;
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
            panNorth();
            return true;
        } else if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
            panSouth();
            return true;
        }
        return false;
    }
```

```
public void panWest() {
   GeoPoint pt = new GeoPoint(mMapView.getMapCenter().getLatitudeE6(),
            mMapView.getMapCenter().getLongitudeE6()
                    - mMapView.getLongitudeSpan() / 4);
    mc.setCenter(pt);
}
public void panEast() {
   GeoPoint pt = new GeoPoint(mMapView.getMapCenter().getLatitudeE6(),
            mMapView.getMapCenter().getLongitudeE6()
                    + mMapView.getLongitudeSpan() / 4);
    mc.setCenter(pt);
}
public void panNorth() {
   GeoPoint pt = new GeoPoint(mMapView.getMapCenter().getLatitudeE6()
            + mMapView.getLatitudeSpan() / 4, mMapView.getMapCenter()
            .getLongitudeE6());
    mc.setCenter(pt);
}

public void panSouth() {
    GeoPoint pt = new GeoPoint(mMapView.getMapCenter().getLatitudeE6()
            - mMapView.getLatitudeSpan() / 4, mMapView.getMapCenter()
            .getLongitudeE6());
    mc.setCenter(pt);
}
public void zoomIn() {
    mc.zoomIn();
}
public void zoomOut() {
    mc.zoomOut();
}
public void toggleSatellite() {
    mMapView.setSatellite(true);
    mMapView.setStreetView(false);
    mMapView.setTraffic(false);
}
public void toggleTraffic() {
    mMapView.setTraffic(true);
    mMapView.setSatellite(false);
    mMapView.setStreetView(false);
}
public void toggleStreetView() {
    mMapView.setStreetView(true);
    mMapView.setSatellite(false);
    mMapView.setTraffic(false);
}
private void centerOnGPSPosition() {
    Log.d(TAG, "centerOnGPSPosition");
    String provider = "gps";
    LocationManager lm = (LocationManager)
        getSystemService(Context.LOCATION_SERVICE);
```

```
        Location loc = lm.getLastKnownLocation(provider);
        loc = lm.getLastKnownLocation(provider);

        mDefPoint = new GeoPoint((int) (loc.getLatitude() * 1000000),
                (int) (loc.getLongitude() * 1000000));
        mDefCaption = "I'm Here.";
        mc.animateTo(mDefPoint);
        mc.setCenter(mDefPoint);
        MyOverlay mo = new MyOverlay();
        mo.onTap(mDefPoint, mMapView);
        mMapView.getOverlays().add(mo);
    }

    protected class MyOverlay extends Overlay {
        @Override
        public void draw(Canvas canvas, MapView mv, boolean shadow) {
            Log.d(TAG, "MyOverlay::darw..mDefCaption=" + mDefCaption);
            super.draw(canvas, mv, shadow);
            if (mDefCaption.length() == 0) {
                return;
            }
            Paint p = new Paint();
            int[] scoords = new int[2];
            int sz = 5;
            Point myScreenCoords = new Point();
            mMapView.getProjection().toPixels(mDefPoint, myScreenCoords);
            scoords[0] = myScreenCoords.x;
            scoords[1] = myScreenCoords.y;
            p.setTextSize(14);
            p.setAntiAlias(true);

            int sw = (int) (p.measureText(mDefCaption) + 0.5f);
            int sh = 25;
            int sx = scoords[0] - sw / 2 - 5;
            int sy = scoords[1] - sh - sz - 2;
            RectF rec = new RectF(sx, sy, sx + sw + 10, sy + sh);
            p.setStyle(Style.FILL);
            p.setARGB(128, 255, 0, 0);
            canvas.drawRoundRect(rec, 5, 5, p);
            p.setStyle(Style.STROKE);
            p.setARGB(255, 255, 255, 255);
            canvas.drawRoundRect(rec, 5, 5, p);

            canvas.drawText(mDefCaption, sx + 5, sy + sh - 8, p);
            p.setStyle(Style.FILL);
            p.setARGB(88, 255, 0, 0);
            p.setStrokeWidth(1);
           RectF spot = new RectF(scoords[0] - sz, scoords[1] + sz, scoords[0]
                   + sz, scoords[1] - sz);
            canvas.drawOval(spot, p);
```



```
            p.setARGB(255, 255, 0, 0);
            p.setStyle(Style.STROKE);
            canvas.drawCircle(scoords[0], scoords[1], sz, p);
        }
    }

        @Override
        public void onLocationChanged(Location loc) {
            Log.d(TAG, "MyLocationListener::onLocationChanged..");
            if (loc != null) {
                Toast.makeText(
                        getBaseContext(),
                        "Location changed : Lat: " + loc.getLatitude()
                                + " Lng: " + loc.getLongitude(),
                        Toast.LENGTH SHORT).show();
               mDefPoint = new GeoPoint((int) (loc.getLatitude() * 1000000),
                        (int) (loc.getLongitude() * 1000000));
                mc.animateTo(mDefPoint);
                mc.setCenter(mDefPoint);
                mDefCaption = "Lat: " + loc.getLatitude() + ",Lng: "
                        + loc.getLongitude();
                MyOverlay mo = new MyOverlay();
                mo.onTap(mDefPoint, mMapView);
                mMapView.getOverlays().add(mo);
                   loc.getLongitude(),loc.getLatitude(), loc.getAltitude());
            }
        }

        @Override
        public void onProviderDisabled(String provider) {
            Toast.makeText(
                    getBaseContext(),
                    "ProviderDisabled.",
                    Toast.LENGTH SHORT).show();     }

        public void onProviderEnabled(String provider) {
            Toast.makeText(
                    getBaseContext(),
                    "ProviderEnabled,provider:"+provider,
                    Toast.LENGTH SHORT).show();     }
        public void onStatusChanged(String provider, int status, Bundle extras) {
        }
    }

    // 初始化菜单
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        menu.add(0, MENU CON, 0, R.string.menu con).setIcon(
                R.drawable.con track).setAlphabeticShortcut('C');
        menu.add(0, MENU DEL, 0, R.string.menu del).setIcon(R.drawable.delete)
                .setAlphabeticShortcut('D');
```

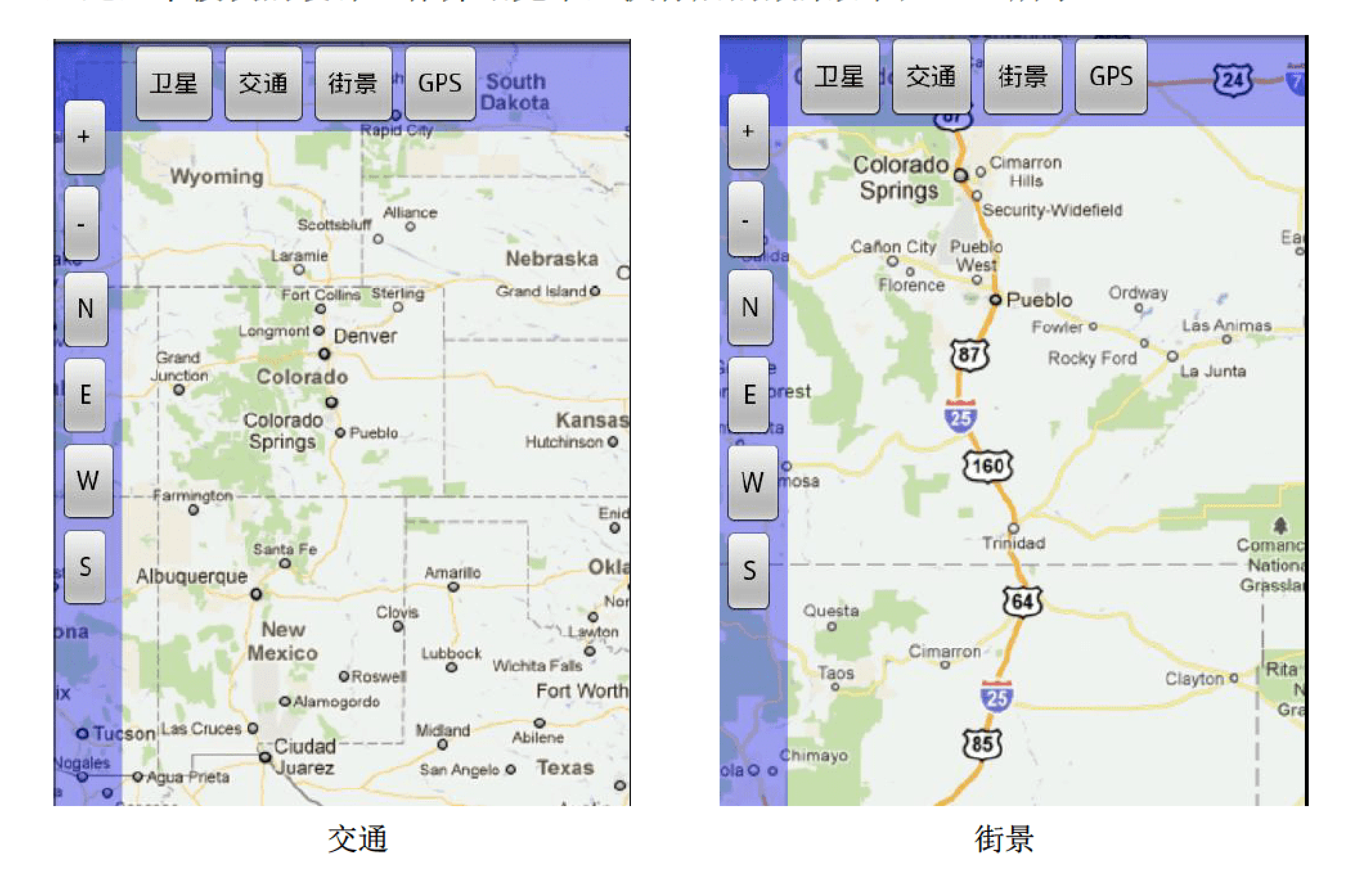
```
        menu.add(0, MENU NEW, 0, R.string.menu new).setIcon(
                R.drawable.new track).setAlphabeticShortcut('N');
        menu.add(0, MENU MAIN, 0, R.string.menu main).setIcon(R.drawable.icon)
                .setAlphabeticShortcut('M');
        return true;
    }
    // 当一个菜单被选中的时候调用
    public boolean onOptionsItemSelected(MenuItem item) {
        Intent intent = new Intent();
        switch (item.getItemId()) {
        case MENU NEW:
            intent.setClass(ShowTrack.this, NewTrack.class);
            startActivity(intent);
            return true;
        case MENU CON:
            // TODO: 继续跟踪选择的记录
            startTrackService();
            return true;
        case MENU DEL:
             mDbHelper = new TrackDbAdapter(this);
             mDbHelper.open();
            if (mDbHelper.deleteTrack(rowId)) {
                mDbHelper.close();
                intent.setClass(ShowTrack.this, iTracks.class);
                startActivity(intent);
            }else{
                mDbHelper.close();
            }
            return true;
        case MENU MAIN:
            intent.setClass(ShowTrack.this, iTracks.class);
            startActivity(intent);
            break;
        }
        return true;
    }

    @Override
    protected void onStop() {
        super.onStop();
        Log.d(TAG, "onStop");
    }

    @Override
   public void onDestroy() {
        Log.d(TAG, "onDestroy.");
       super.onDestroy();
       stopTrackService();
       }
}
```



至此，本模块的设计工作介绍完毕，执行后的效果如图 11-11 所示。



交通　　　　街景

图 11-11　地图展示界面

11.2.7　数据存取

在前面介绍的系统需求分析中，系统要求将每次跟踪的目标位置保存在数据库中，并且每次改变后都要保存起来。本项目的个性化配置信息保存在 SharedPreferences 中，在此将需要存取的数据放在数据库中。在 Android 中，存取数据库的方法有两种：一种是通过 help 类继承 SQLiteDatabase 相关类绑定 SQL，另外一种是使用 ContentProvideer 进行封装。

1. 创建数据库

本项目需要同时操作数据库中的两个表。在此先在文件 DbAdapter.java 中创建一个名为 DbAdapter 的类，并重新定义了 SQLiteOpenHelper 的 onCreate 方法和 onUpgrade 方法，通过这两个方法实现了创建和升级数据库的脚本。文件 DbAdapter.java 的实现代码如下。

```
package com.iceskysl.map;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;
public class DbAdapter {
    private static final String TAG = "DbAdapter";
    private static final String DATABASE NAME = "iTracks.db";
    private static final int DATABASE VERSION = 1;
```

```
    public class DatabaseHelper extends SQLiteOpenHelper {
        public DatabaseHelper(Context context) {
            super(context, DATABASE NAME, null, DATABASE VERSION);
        }
        @Override
        public void onCreate(SQLiteDatabase db) {
            String tracks sql = "CREATE TABLE " + TrackDbAdapter.TABLE NAME + " ("
            + TrackDbAdapter.ID + " INTEGER primary key autoincrement, "
            + TrackDbAdapter.NAME   + " text not null, "
            + TrackDbAdapter.DESC + " text ,"
            + TrackDbAdapter.DIST + " LONG ,"
            + TrackDbAdapter.TRACKEDTIME + " LONG ,"
            + TrackDbAdapter.LOCATE COUNT + " INTEGER, "
            + TrackDbAdapter.CREATED + " text, "
            + TrackDbAdapter.AVGSPEED + " LONG, "
            + TrackDbAdapter.MAXSPEED + " LONG ,"
            + TrackDbAdapter.UPDATED + " text "
            + ");";
            Log.i(TAG, tracks sql);
            db.execSQL(tracks sql);

            String locats sql = "CREATE TABLE " + LocateDbAdapter.TABLE NAME + " ("
            + LocateDbAdapter.ID+ " INTEGER primary key autoincrement, "
            + LocateDbAdapter.TRACKID   + " INTEGER not null, "
            + LocateDbAdapter.LON + " DOUBLE ,"
            + LocateDbAdapter.LAT + " DOUBLE ,"
            + LocateDbAdapter.ALT + " DOUBLE ,"
            + LocateDbAdapter.CREATED + " text "
            + ");";
            Log.i(TAG, locats sql);
            db.execSQL(locats sql);
        }
        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
           db.execSQL("DROP TABLE IF EXISTS " + LocateDbAdapter.TABLE NAME + ";");
           db.execSQL("DROP TABLE IF EXISTS " + TrackDbAdapter.TABLE NAME + ";");
            onCreate(db);
        }
    }

}
```

2. 操作数据库

通过对数据库的操作实现了对两个表操作的封装处理，因为共用了同一个数据库，所以只需从前面创建的 bAdapter 中继续继承即可，在此继承出了两个类：TrackDbAdapter 和 LocateDbAdapter。通过对这两个类的封装，实现了对数据表的操作。

(1) 类 TrackDbAdapter 是在文件 TrackDbAdapter.java 中定义的，在此先声明了一些常量，然后根据需要的操作功能定义了具体方法。其代码如下。



```
package com.iceskysl.map;

import java.util.Calendar;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.util.Log;

public class TrackDbAdapter extends DbAdapter{
    private static final String TAG = "TrackDbAdapter";

    public static final String TABLE NAME = "tracks";

    public static final String ID = " id";
    public static final String KEY ROWID = " id";
    public static final String NAME = "name";
    public static final String DESC = "desc";
    public static final String DIST = "distance";
    public static final String TRACKEDTIME = "tracked time";
    public static final String LOCATE COUNT = "locats count";
    public static final String CREATED = "created at";
    public static final String UPDATED = "updated at";
    public static final String AVGSPEED = "avg speed";
    public static final String MAXSPEED = "max speed";

    private DatabaseHelper mDbHelper;
    private SQLiteDatabase mDb;
    private final Context mCtx;

    public TrackDbAdapter(Context ctx) {
        this.mCtx = ctx;
    }

    public TrackDbAdapter open() throws SQLException {
        mDbHelper = new DatabaseHelper(mCtx);
        mDb = mDbHelper.getWritableDatabase();
        return this;
    }

    public void close() {
        mDbHelper.close();
    }
    public Cursor getTrack(long rowId) throws SQLException {
        Cursor mCursor =
        mDb.query(true, TABLE NAME, new String[] { KEY ROWID, NAME,
                DESC, CREATED }, KEY ROWID + "=" + rowId, null, null,
                null, null, null);
        if (mCursor != null) {
```

```
            mCursor.moveToFirst();
        }
        return mCursor;
    }

    public long createTrack(String name, String desc) {
        Log.d(TAG, "createTrack.");
        ContentValues initialValues = new ContentValues();
        initialValues.put(NAME, name);
        initialValues.put(DESC, desc);
        Calendar calendar = Calendar.getInstance();
        String created = calendar.get(Calendar.YEAR) + "-" +calendar.get
            (Calendar.MONTH) + "-" + calendar.get(Calendar.DAY OF MONTH) + " "
            + calendar.get(Calendar.HOUR OF DAY) + ":"
            + calendar.get(Calendar.MINUTE) + ":" + calendar.get(Calendar.SECOND);
        initialValues.put(CREATED, created);
        initialValues.put(UPDATED, created);
        return mDb.insert(TABLE NAME, null, initialValues);
    }

    //
    public boolean deleteTrack(long rowId) {
       return mDb.delete(TABLE NAME, KEY ROWID + "=" + rowId, null) > 0;
    }

    public Cursor getAllTracks() {
        return mDb.query(TABLE NAME, new String[] { ID, NAME,
               DESC, CREATED }, null, null, null, null, "updated at desc");
    }

    public boolean updateTrack(long rowId, String name, String desc) {
        ContentValues args = new ContentValues();
        args.put(NAME, name);
        args.put(DESC, desc);
        Calendar calendar = Calendar.getInstance();
        String updated = calendar.get(Calendar.YEAR) + "-" +calendar.get
            (Calendar.MONTH) + "-" + calendar.get(Calendar.DAY OF MONTH) + " "
            + calendar.get(Calendar.HOUR OF DAY) + ":"
            + calendar.get(Calendar.MINUTE) + ":" + calendar.get(Calendar.SECOND);
        args.put(UPDATED, updated);
       return mDb.update(TABLE NAME, args, KEY ROWID + "=" + rowId, null) > 0;
    }

}
```

(2) 类 LocateDbAdapter 是在文件 LocateDbAdapter.java 中实现的，在此也是首先声明了一些常量，然后根据需要的操作功能定义了具体方法。其代码如下。

```
package com.iceskysl.map;

import java.util.Calendar;
```

```
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.util.Log;

public class LocateDbAdapter extends DbAdapter {
    private static final String TAG = "LocateDbAdapter";
    public static final String TABLE NAME = "locates";

    public static final String ID = " id";
    public static final String TRACKID = "track id";
    public static final String LON = "longitude";
    public static final String LAT = "latitude";
    public static final String ALT = "altitude";
    public static final String CREATED = "created at";

    private DatabaseHelper mDbHelper;
    private SQLiteDatabase mDb;
    private final Context mCtx;

    public LocateDbAdapter(Context ctx) {
        this.mCtx = ctx;
    }

    public LocateDbAdapter open() throws SQLException {
        mDbHelper = new DatabaseHelper(mCtx);
        mDb = mDbHelper.getWritableDatabase();
        return this;
    }

    public void close() {
        mDbHelper.close();
    }

    public Cursor getLocate(long rowId) throws SQLException {
        Cursor mCursor =
        mDb.query(true, TABLE NAME, new String[] { ID, LON,
                LAT, ALT, CREATED }, ID + "=" + rowId, null, null,
                null, null, null);
        if (mCursor != null) {
            mCursor.moveToFirst();
        }
        return mCursor;
    }

    public long createLocate(int track id, Double longitude, Double latitude,
        Double altitude) {
        Log.d(TAG, "createLocate.");
        ContentValues initialValues = new ContentValues();
```

```
        initialValues.put(TRACKID, track id);
        initialValues.put(LON, longitude);
        initialValues.put(LAT, latitude);
        initialValues.put(ALT, altitude);

        Calendar calendar = Calendar.getInstance();
        String created = calendar.get(Calendar.YEAR) + "-" +calendar.get
            (Calendar.MONTH) + "-" + calendar.get(Calendar.DAY OF MONTH) + " "
            + calendar.get(Calendar.HOUR OF DAY) + ":"
            + calendar.get(Calendar.MINUTE) + ":"  + calendar.get
              (Calendar.SECOND);
        initialValues.put(CREATED, created);
        return mDb.insert(TABLE NAME, null, initialValues);
    }
    public boolean deleteLocate(long rowId) {
        return mDb.delete(TABLE NAME, ID + "=" + rowId, null) > 0;
    }


    public Cursor getTrackAllLocates(int trackId) {
        return mDb.query(TABLE NAME, new String[] { ID,TRACKID, LON,
                LAT, ALT,CREATED }, "track id=?", new String[]
                {String.valueOf(trackId)}, null, null, "created at asc");
    }
}
```

11.2.8 实现 Service 服务

本项目要求在切换一个界面的时候不会影响到对目标的追踪。即使来到另外一个新界面，程序也需要在后台进行跟踪和记录。根据上述要求，决定了本项目需要用到 Service 服务。

首先在文件 AndroidManifest.xml 中加入对 Service 的声明，在此添加一个名为 Track 的 Service，并设定了其名字为“com.iceskysl.map.START_TRACK_SERVICE”。具体代码如下。

```
        <service android:name=".Track">
            <intent-filter>
                <action android:name="com.iceskysl.map.START TRACK SERVICE" />
                <category android:name="android.intent.category.default" />
            </intent-filter>
        </service>
```

再看处理文件 Track.java，在此设置 Track 继承于 Service 类，然后在 onStart 中连接了数据库，接收了参数并设定了监听器，并使用了 MyLocationListener，当位置变化(onLocationChanged)的时候，调用前面数据存储部分已经实现的 mlcDbHelper.createLocate 方法，将位置信息和接收到的参数写入到数据库中。文件 Track.java 的主要代码如下。

```
public class Track extends Service {
    private static final String TAG = "Track";
```



```
private LocationManager lm;
private LocationListener locationListener;

static LocateDbAdapter mlcDbHelper = null;
private int track id;

@Override
public IBinder onBind(Intent arg0) {
    Log.d(TAG, "onBind.");
    return null;
}

public void onStart(Intent intent, int startId) {
    Log.d(TAG, "onStart.");
   super.onStart(intent, startId);
   startDb();
    Bundle extras = intent.getExtras();
    if (extras != null) {
        track id = extras.getInt(LocateDbAdapter.TRACKID);
    }
    Log.d(TAG, "track id =" + track id);
    // ---use the LocationManager class to obtain GPS locations---
    lm = (LocationManager) getSystemService(Context.LOCATION SERVICE);
    locationListener = new MyLocationListener();
    lm.requestLocationUpdates(LocationManager.GPS PROVIDER, 0, 0,
            locationListener);
}

private void startDb() {
    if(mlcDbHelper == null){
       mlcDbHelper = new LocateDbAdapter(this);
        mlcDbHelper.open();
    }
}

private void stopDb() {
    if(mlcDbHelper != null){
       mlcDbHelper.close();
    }
}


public static LocateDbAdapter getDbHelp(){
    return mlcDbHelper;
}


public void onDestroy() {
    Log.d(TAG, "onDestroy.");
   super.onDestroy();
```

```
        stopDb();
    }
    protected class MyLocationListener implements LocationListener {

        @Override
        public void onLocationChanged(Location loc) {
            Log.d(TAG, "MyLocationListener::onLocationChanged..");
            if (loc != null) {
                if(mlcDbHelper == null){
                    mlcDbHelper.open();
                }
                mlcDbHelper.createLocate(track id, loc.getLongitude(),
                    loc.getLatitude(), loc.getAltitude());
            }
        }

        @Override
        public void onProviderDisabled(String provider) {
            Toast.makeText(
                    getBaseContext(),
                    "ProviderDisabled.",
                    Toast.LENGTH SHORT).show();     }

        @Override
        public void onProviderEnabled(String provider) {
            Toast.makeText(
                    getBaseContext(),
                    "ProviderEnabled,provider:"+provider,
                    Toast.LENGTH SHORT).show();     }
        @Override
        public void onStatusChanged(String provider, int status, Bundle extras) {
            // TODO Auto-generated method stub
        }
    }
}
```

11.3 发布自己的作品来盈利

当一个 Android 项目开发完毕后，需要打包和签名处理，这样才能放到手机中使用，当然也可以发布到 Market 上去赚钱。下面开始讲解打包、签名、发布 Android 程序的具体过程。

11.3.1 申请会员

申请会员即去 Market 申请成为会员，具体流程如下。

(1) 登录 http://market.android/publish/signup，如图 11-12 所示。

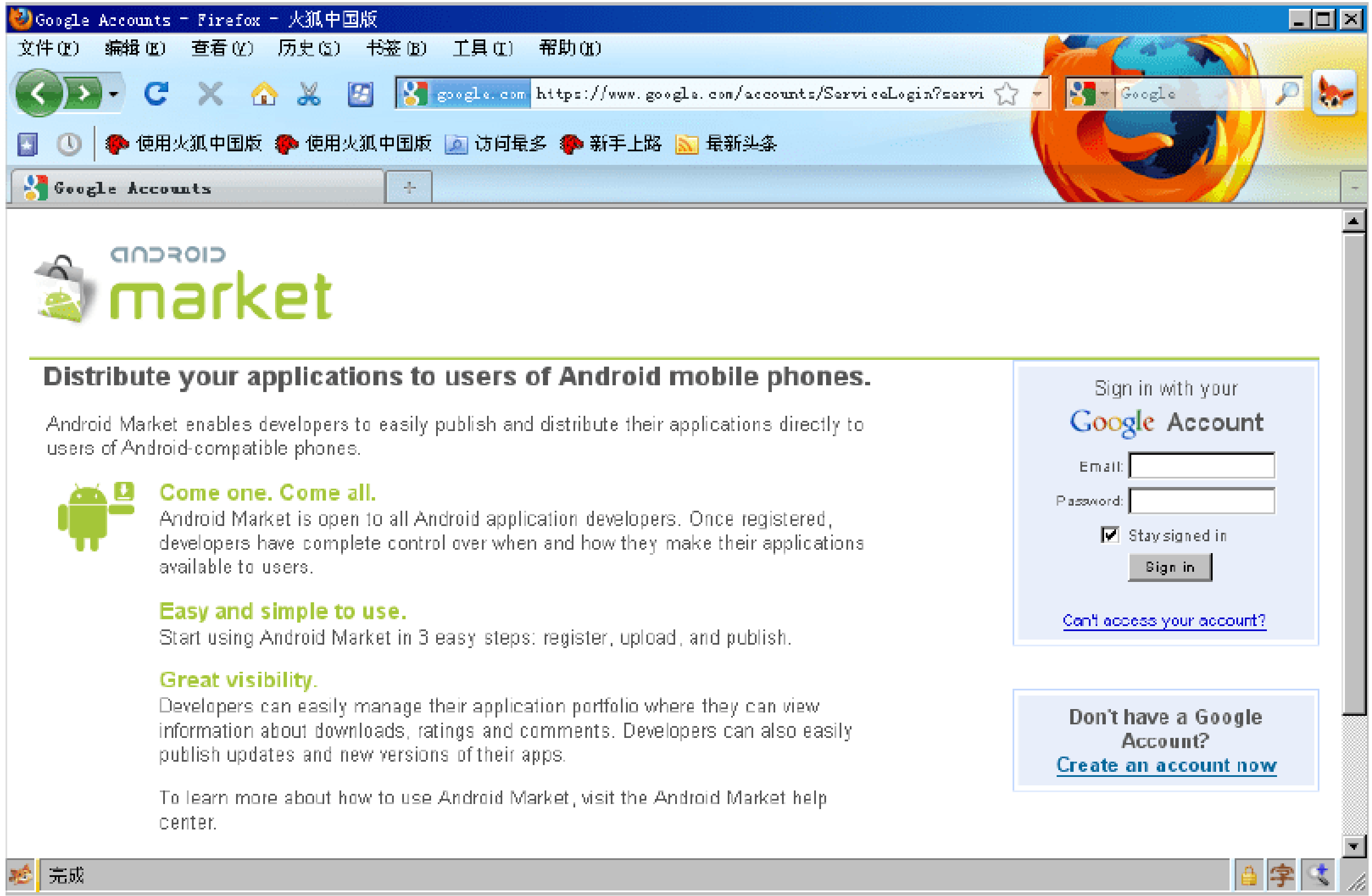


图 11-12　登录 Market

(2) 单击链接 Create an account now，来到注册页面，如图 11-13 所示。

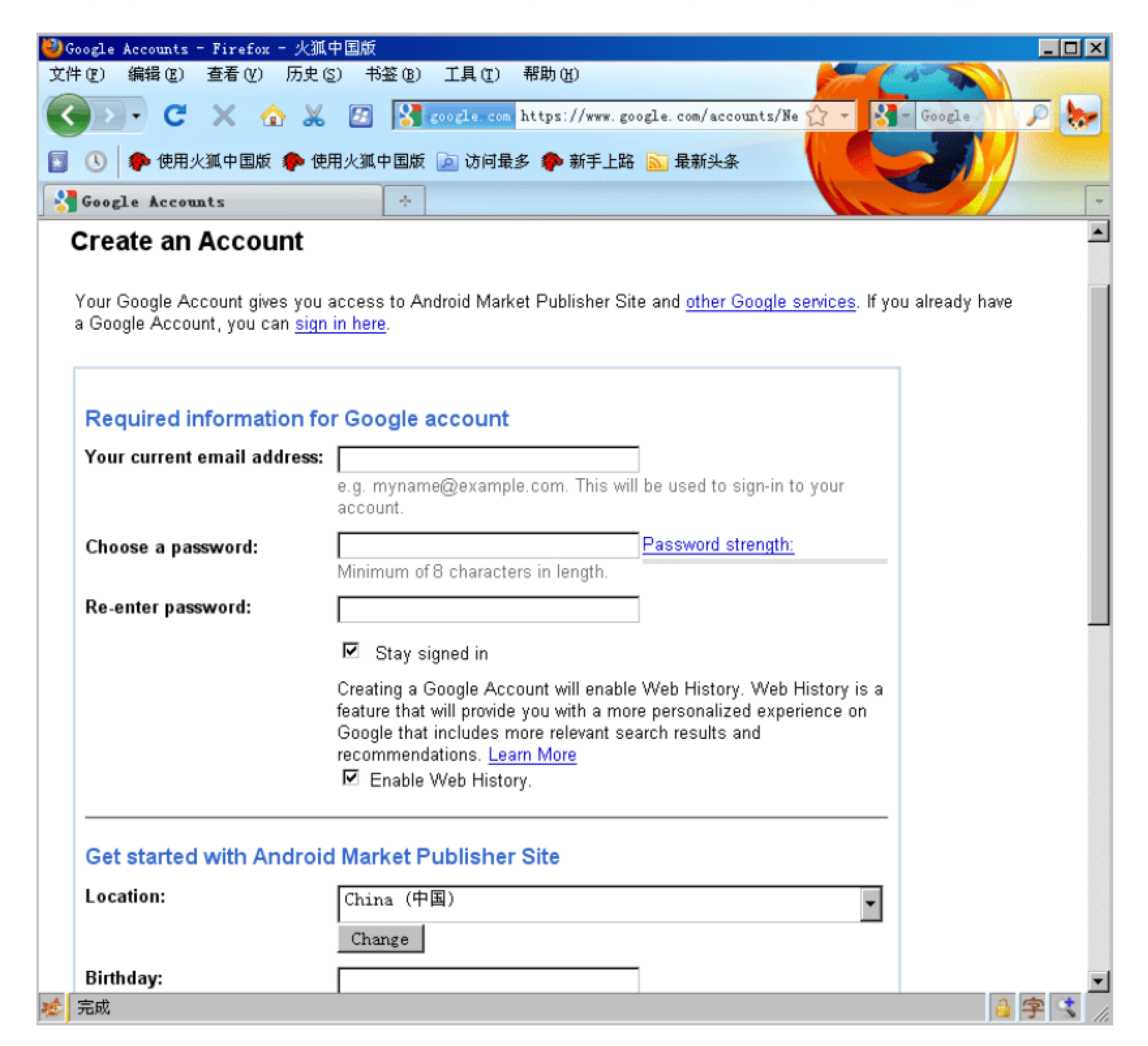


图 11-13　注册页面

(3) 单击同意协议后来到下一页面，在此输入手机号码，如图 11-14 所示。

(4) 在新页面中输入手机获取的验证码，如图 11-15 所示。

(5) 验证通过后，在新页面中继续输入信息，如图 11-16 所示。

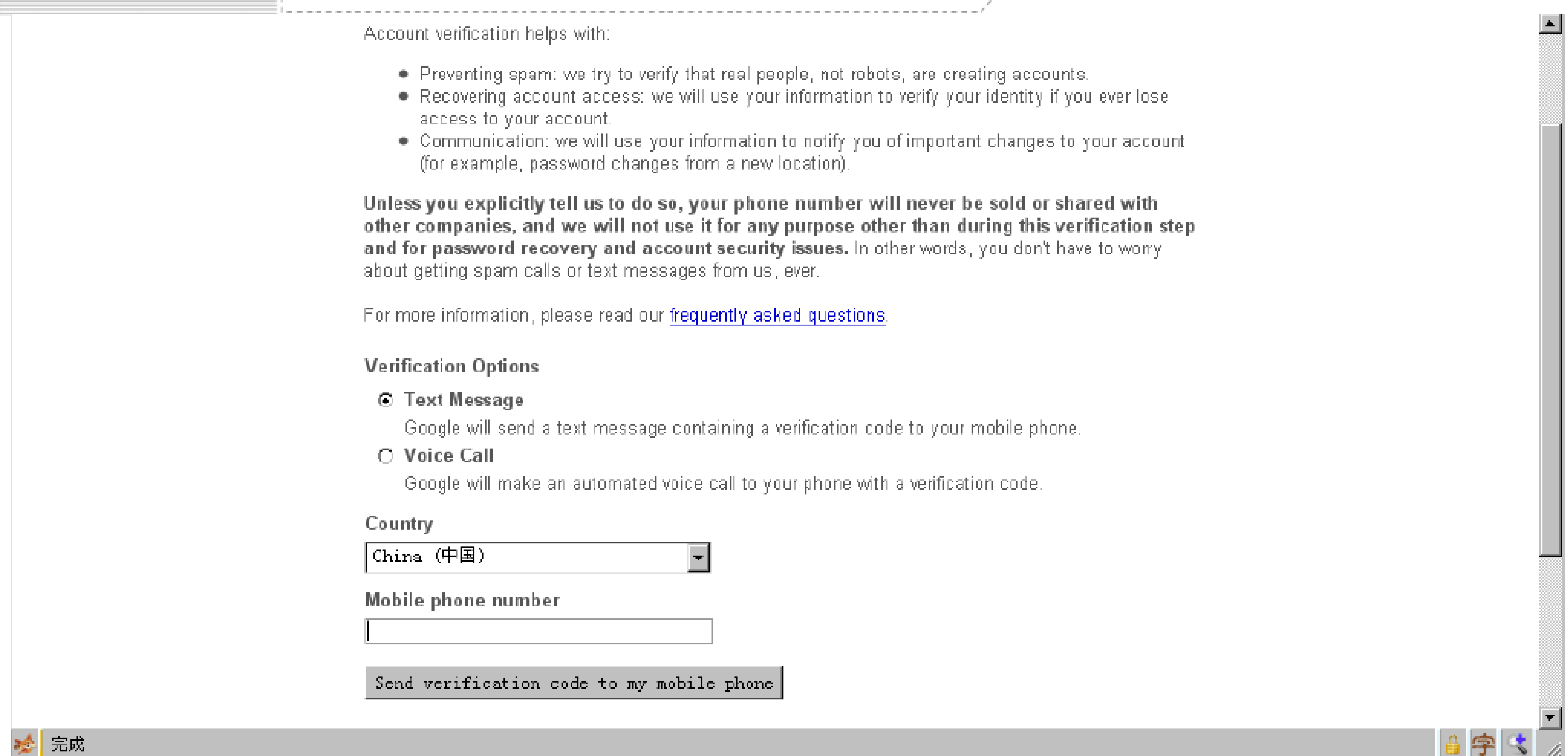


图 11-14　输入手机号码

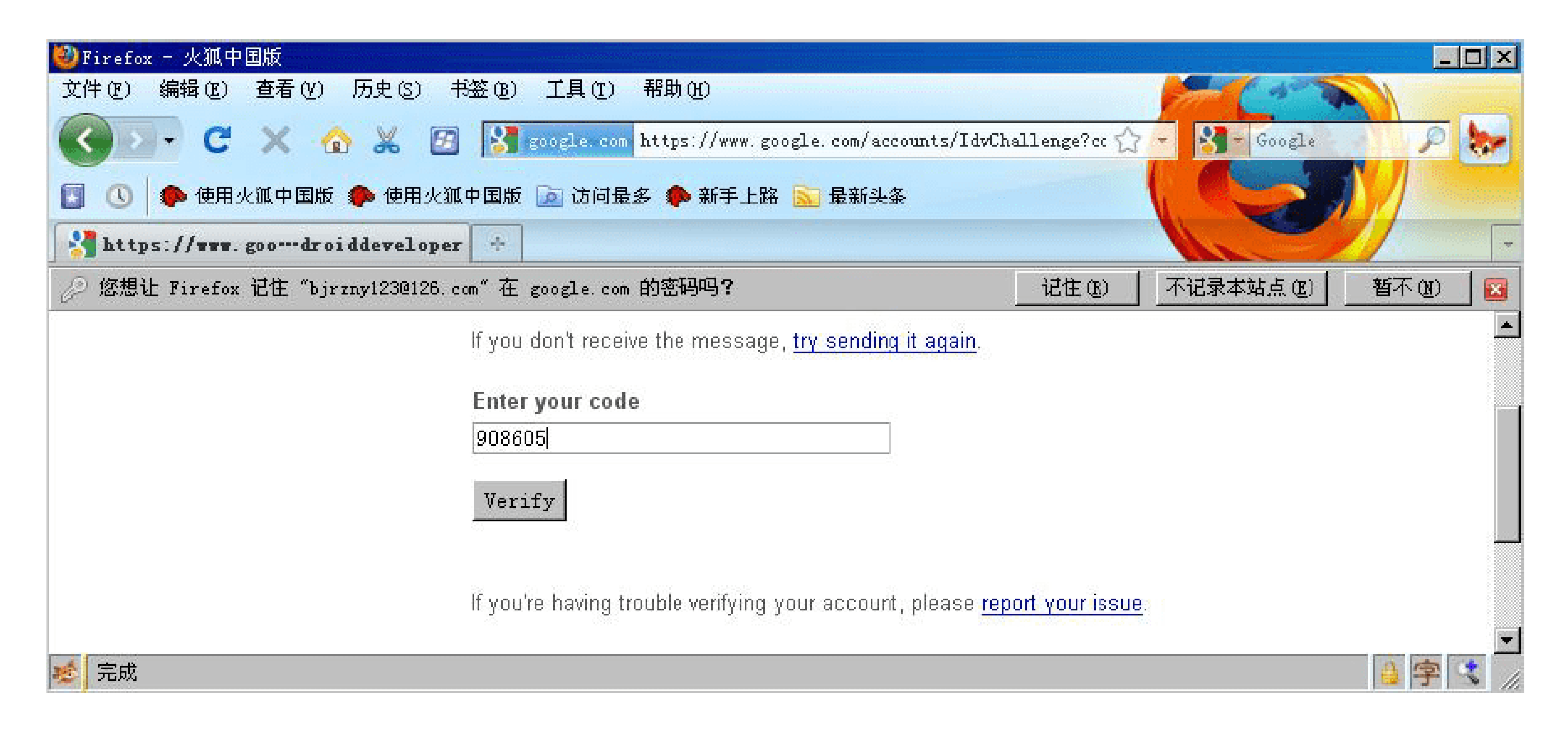


图 11-15　输入验证码

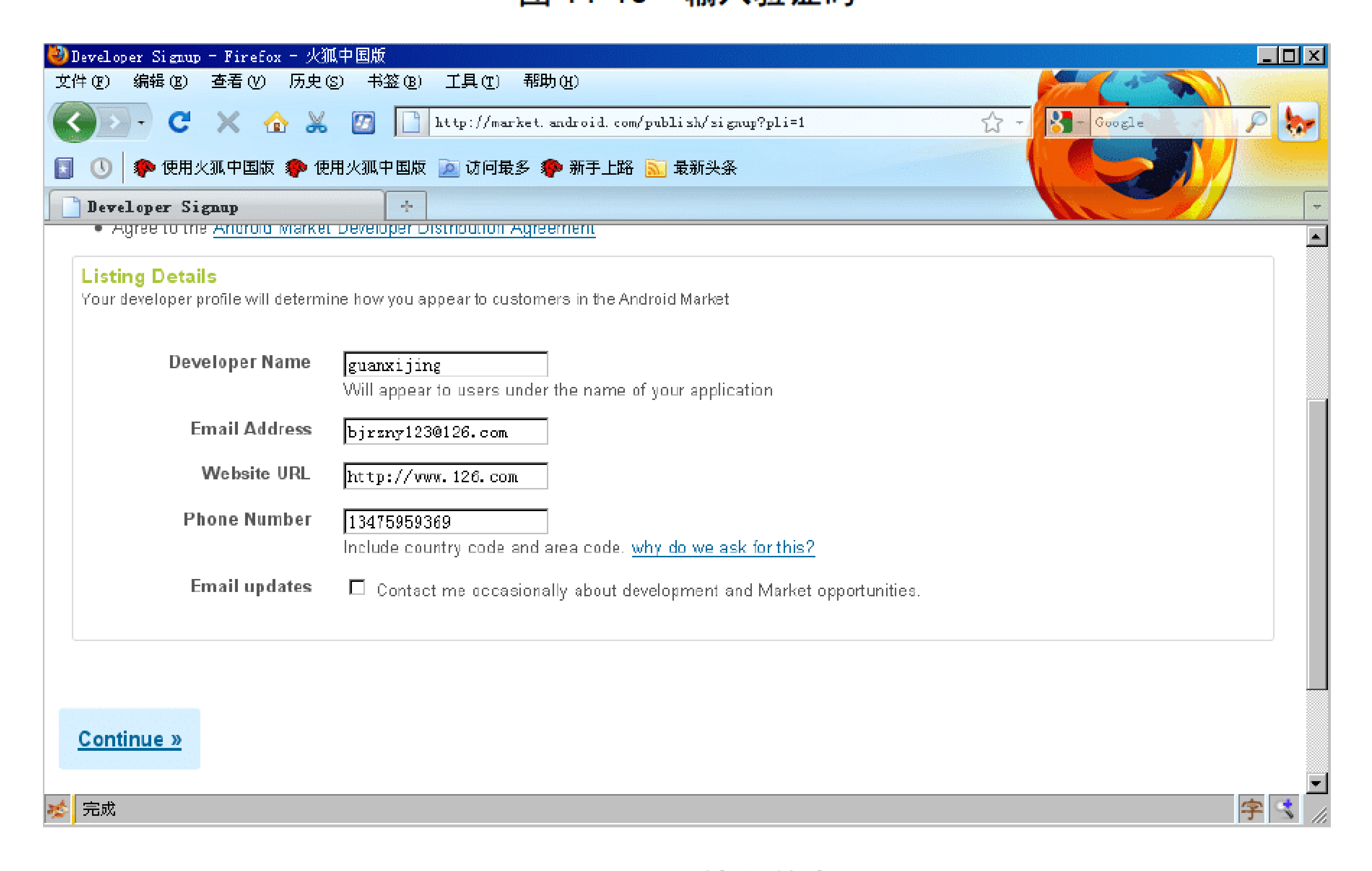


图 11-16　输入信息

(6) 单击 Continue 按钮后，提示需要花费 25 美元，支付后才能成为正式会员，如图 11-17 所示。

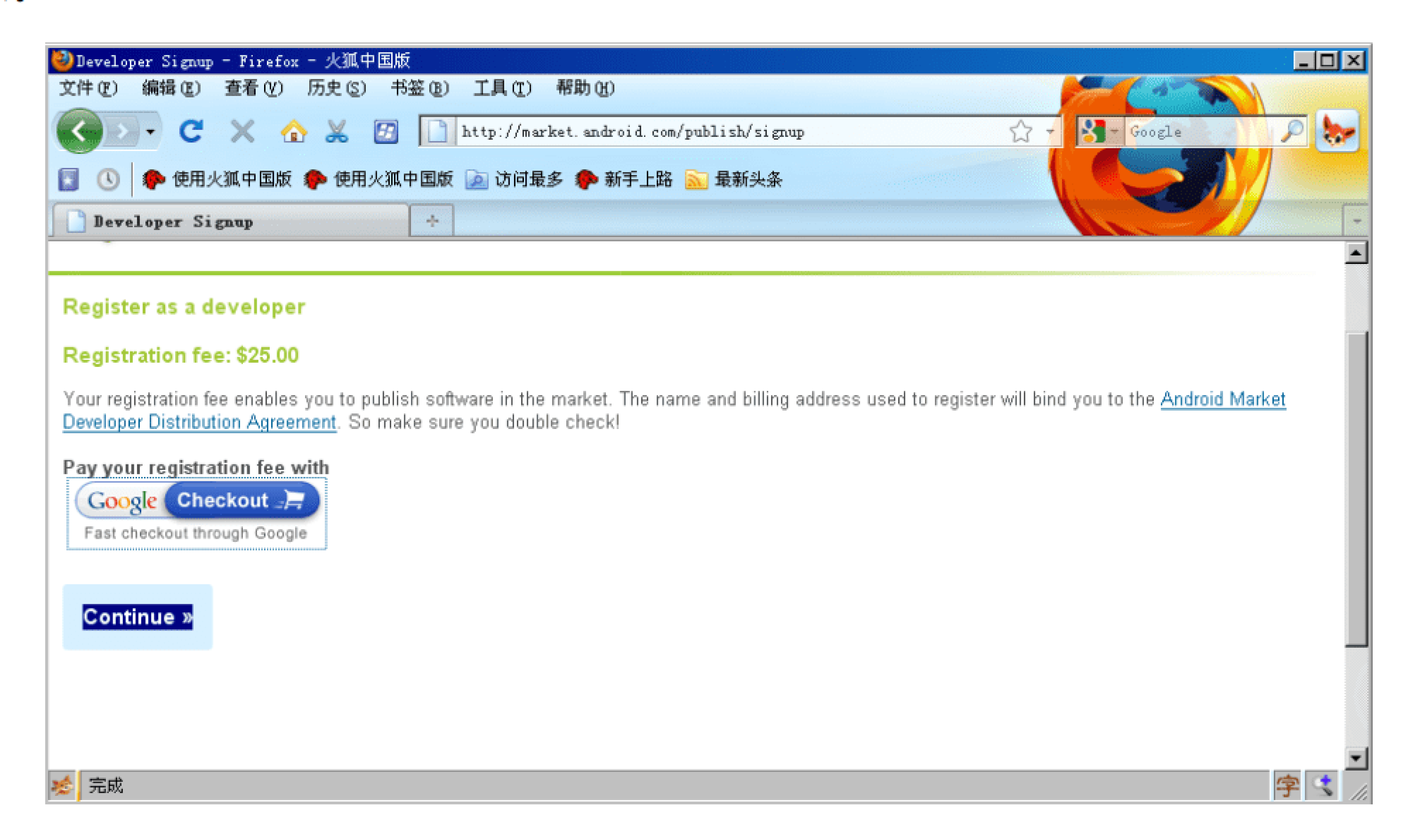


图 11-17　需要支付界面

(7) 单击 Google Checkout Fast checkout through Google 按钮来到支付界面，如图 11-18 所示。

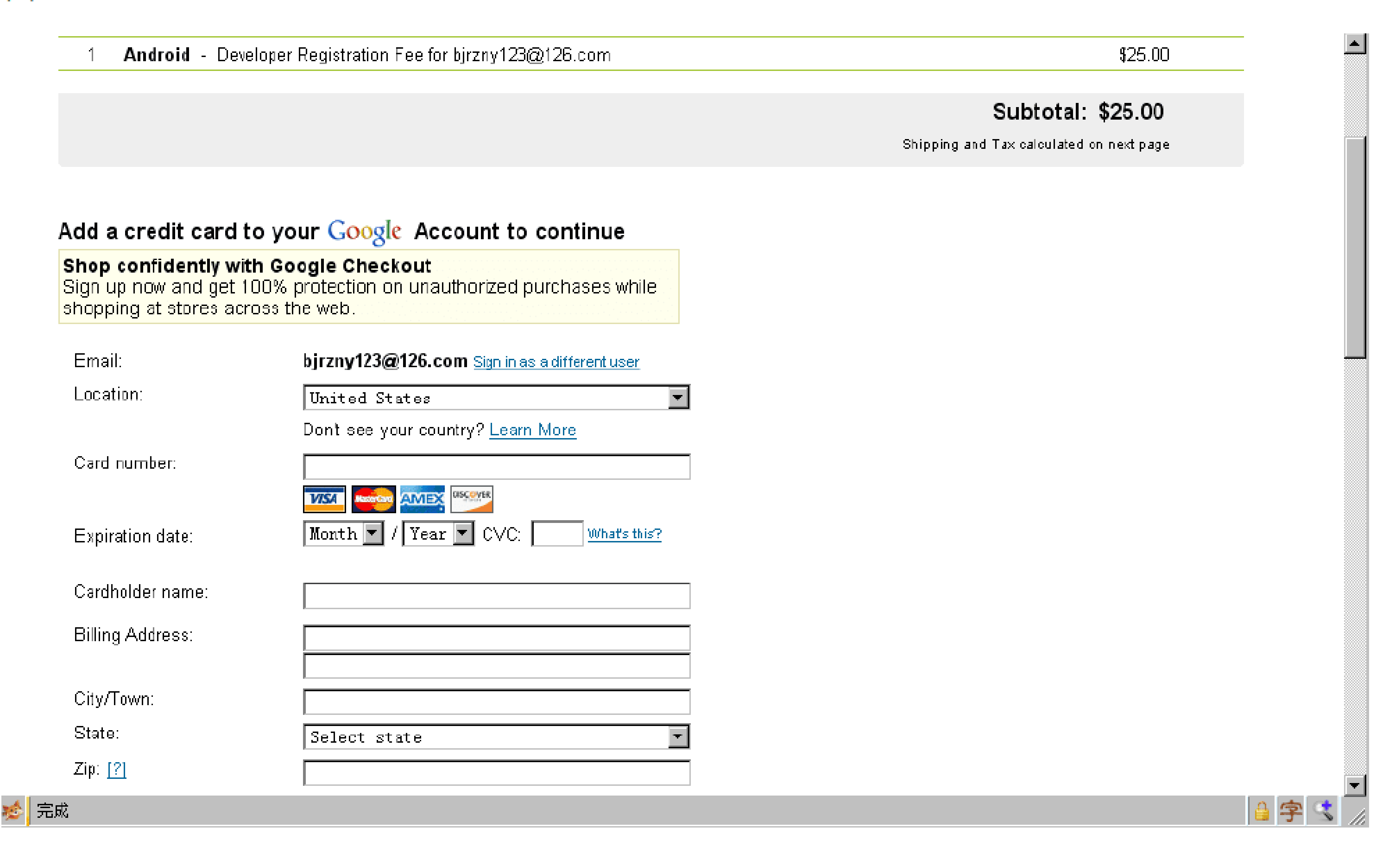


图 11-18　支付界面

在此输入你的信用卡信息，完成支付后即可成为正式会员。

11.3.2　生成签名文件

Android 程序的签名和 Symbian 类似，都可以自签名(Self-signed)，但是在 Android 平

台中证书初期还显得形同虚设，平时开发时通过 ADB 接口上传的程序会自动被签有 Debug 权限的程序。需要签名验证再上传程序到 Android Market 上时，大家都已经发现这个问题了。

Android 签名文件的制作方法有以下两种。

1) 命令行生成

具体流程如下。

(1) cmd 命令如下：

```
keytool -genkey -alias android123.keystore -keyalg RSA -validity 20000 -
keystore android123.keystore
```

然后依次提示用户输入如下信息。

输入 keystore 密码：[密码不回显]

再次输入新密码：[密码不回显]

您的名字与姓氏是什么？

[Unknown]： android123

您的组织单位名称是什么？

[Unknown]： www.android123.com.cn

您的组织名称是什么？

[Unknown]： www.android123.com.cn

您的组织名称是什么？

[Unknown]： www.android123.com.cn

您所在的城市或区域名称是什么？

[Unknown]： New York

您所在的州或省份名称是什么？

[Unknown]： New York

该单位的两字母国家代码是什么？

[Unknown]： CN

CN=android123, OU=www.android123.com.cn, O=www.android123.com.cn, L=New York, ST=New York, C=CN 正确吗？

[否]：Y

输入<android123.keystore>的主密码(如果和 keystore 密码相同，按回车)：

其中参数-validity 为证书有效天数，这里我们写的大些：200 天。还有在输入密码时没有回显，只管输入就可以了，一般建议使用 20 位，最后需要记下来后面还要用。接下来就可以为 apk 文件签名了。

(2) 执行

```
jarsigner -verbose -keystore android123.keystore -signedjar
android123_signed.apk android123.apk android123.keystore
```

这样就可以生成签名的 APK 文件，假设输入文件 android123.apk，则最终生成 android123_signed.apk 为 Android 签名后的 APK 执行文件。

注意： keytool 用法和 jarsigner 用法总结。

① keytool 用法：

```
-certreq      [-v] [-protected]
                [-alias <别名>] [-sigalg <sigalg>]
                [-file <csr_file>] [-keypass <密钥库口令>]
                [-keystore <密钥库>] [-storepass <存储库口令>]
                [-storetype <存储类型>] [-providername <名称>]
                [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                [-providerpath <路径列表>]

-changealias [-v] [-protected] -alias <别名> -destalias <目标别名>
                [-keypass <密钥库口令>]
                [-keystore <密钥库>] [-storepass <存储库口令>]
                [-storetype <存储类型>] [-providername <名称>]
                [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                [-providerpath <路径列表>]

-delete       [-v] [-protected] -alias <别名>
                [-keystore <密钥库>] [-storepass <存储库口令>]
                [-storetype <存储类型>] [-providername <名称>]
                [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                [-providerpath <路径列表>]

-exportcert [-v] [-rfc] [-protected]
                [-alias <别名>] [-file <认证文件>]
                [-keystore <密钥库>] [-storepass <存储库口令>]
                [-storetype <存储类型>] [-providername <名称>]
                [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                [-providerpath <路径列表>]

-genkeypair [-v] [-protected]
                [-alias <别名>]
                [-keyalg <keyalg>] [-keysize <密钥大小>]
                [-sigalg <sigalg>] [-dname <dname>]
                [-validity <valDays>] [-keypass <密钥库口令>]
                [-keystore <密钥库>] [-storepass <存储库口令>]
                [-storetype <存储类型>] [-providername <名称>]
                [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                [-providerpath <路径列表>]
```

```
-genseckey     [-v] [-protected]
               [-alias <别名>] [-keypass <密钥库口令>]
               [-keyalg <keyalg>] [-keysize <密钥大小>]
               [-keystore <密钥库>] [-storepass <存储库口令>]
               [-storetype <存储类型>] [-providername <名称>]
               [-providerclass <提供方类名称> [-providerarg <参数>]] ...
               [-providerpath <路径列表>]

-importcert [-v] [-noprompt] [-trustcacerts] [-protected]
               [-alias <别名>]
               [-file <认证文件>] [-keypass <密钥库口令>]
               [-keystore <密钥库>] [-storepass <存储库口令>]
               [-storetype <存储类型>] [-providername <名称>]
               [-providerclass <提供方类名称> [-providerarg <参数>]] ...
               [-providerpath <路径列表>]

-importkeystore [-v]
               [-srckeystore <源密钥库>] [-destkeystore <目标密钥库>]
               [-srcstoretype <源存储类型>] [-deststoretype <目标存储类型>]
               [-srcstorepass <源存储库口令>] [-deststorepass <目标存储库口令>]
               [-srcprotected] [-destprotected]
               [-srcprovidername <源提供方名称>]
               [-destprovidername <目标提供方名称>]
               [-srcalias <源别名> [-destalias <目标别名>]
                 [-srckeypass <源密钥库口令>] [-destkeypass <目标密钥库口令>]]
               [-noprompt]
               [-providerclass <提供方类名称> [-providerarg <参数>]] ...
               [-providerpath <路径列表>]

-keypasswd     [-v] [-alias <别名>]
               [-keypass <旧密钥库口令>] [-new <新密钥库口令>]
               [-keystore <密钥库>] [-storepass <存储库口令>]
               [-storetype <存储类型>] [-providername <名称>]
               [-providerclass <提供方类名称> [-providerarg <参数>]] ...
               [-providerpath <路径列表>]

-list          [-v | -rfc] [-protected]
               [-alias <别名>]
```

[-keystore <密钥库>] [-storepass <存储库口令>]
[-storetype <存储类型>] [-providername <名称>]
[-providerclass <提供方类名称> [-providerarg <参数>]] ...
[-providerpath <路径列表>]

-printcert　[-v] [-file <认证文件>]

-storepasswd [-v] [-new <新存储库口令>]
[-keystore <密钥库>] [-storepass <存储库口令>]
[-storetype <存储类型>] [-providername <名称>]
[-providerclass <提供方类名称> [-providerarg <参数>]] ...
[-providerpath <路径列表>]

② jarsigner 用法:
[选项] jar 文件别名
jarsigner -verify [选项] jar 文件

[-keystore <url>]	密钥库位置
[-storepass <口令>]	用于密钥库完整性的口令
[-storetype <类型>]	密钥库类型
[-keypass <口令>]	专用密钥的口令(如果不同)
[-sigfile <文件>]	.SF/.DSA 文件的名称
[-signedjar <文件>]	已签名的 Jar 文件的名称
[-digestalg <算法>]	摘要算法的名称
[-sigalg <算法>]	签名算法的名称
[-verify]	验证已签名的 Jar 文件
[-verbose]	签名/验证时输出详细信息
[-certs]	输出详细信息和验证时显示证书
[-tsa <url>]	时间戳机构的位置
[-tsacert <别名>]	时间戳机构的公共密钥证书
[-altsigner <类>]	替代的签名机制的类名
[-altsignerpath <路径列表>]	替代的签名机制的位置
[-internalsf]	在签名块内包含 .SF 文件
[-sectionsonly]	不计算整个清单的散列
[-protected]	密钥库已保护验证路径
[-providerName <名称>]	提供者名称
[-providerClass <类>	加密服务提供者的名称
[-providerArg <参数>]] ...	主类文件和构造函数参数

2) 使用 Eclipse ADT 生成

实际上，使用 Eclipse 可以更加直观、方便地生成签名文件，具体流程如下。

(1) 右击 Eclipse 项目名，依次选择 Android Tools | Export Signed Application Package 命令，如图 11-19 所示。

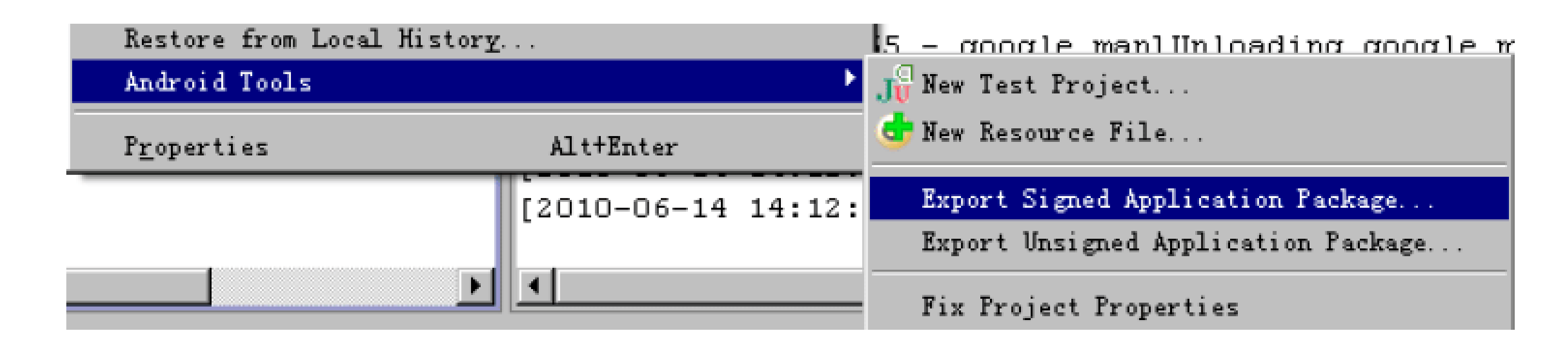


图 11-19 选择导出命令

(2) 在弹出的对话框中选择要导出的项目，如图 11-20 所示。

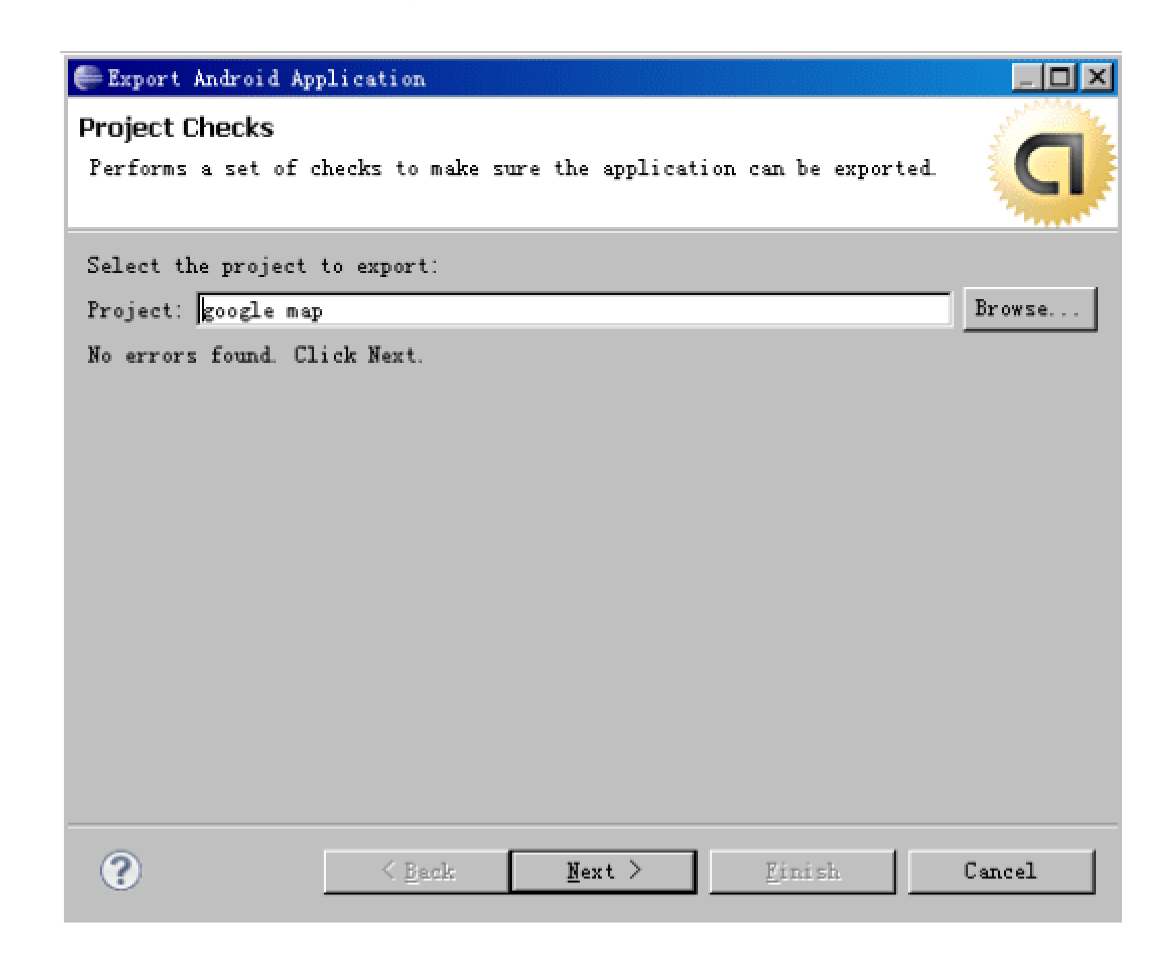


图 11-20 选择要导出的项目

(3) 单击 Next 按钮，在弹出的对话框中选择 Create new keystore，然后分别输入文件名和密码，如图 11-21 所示。

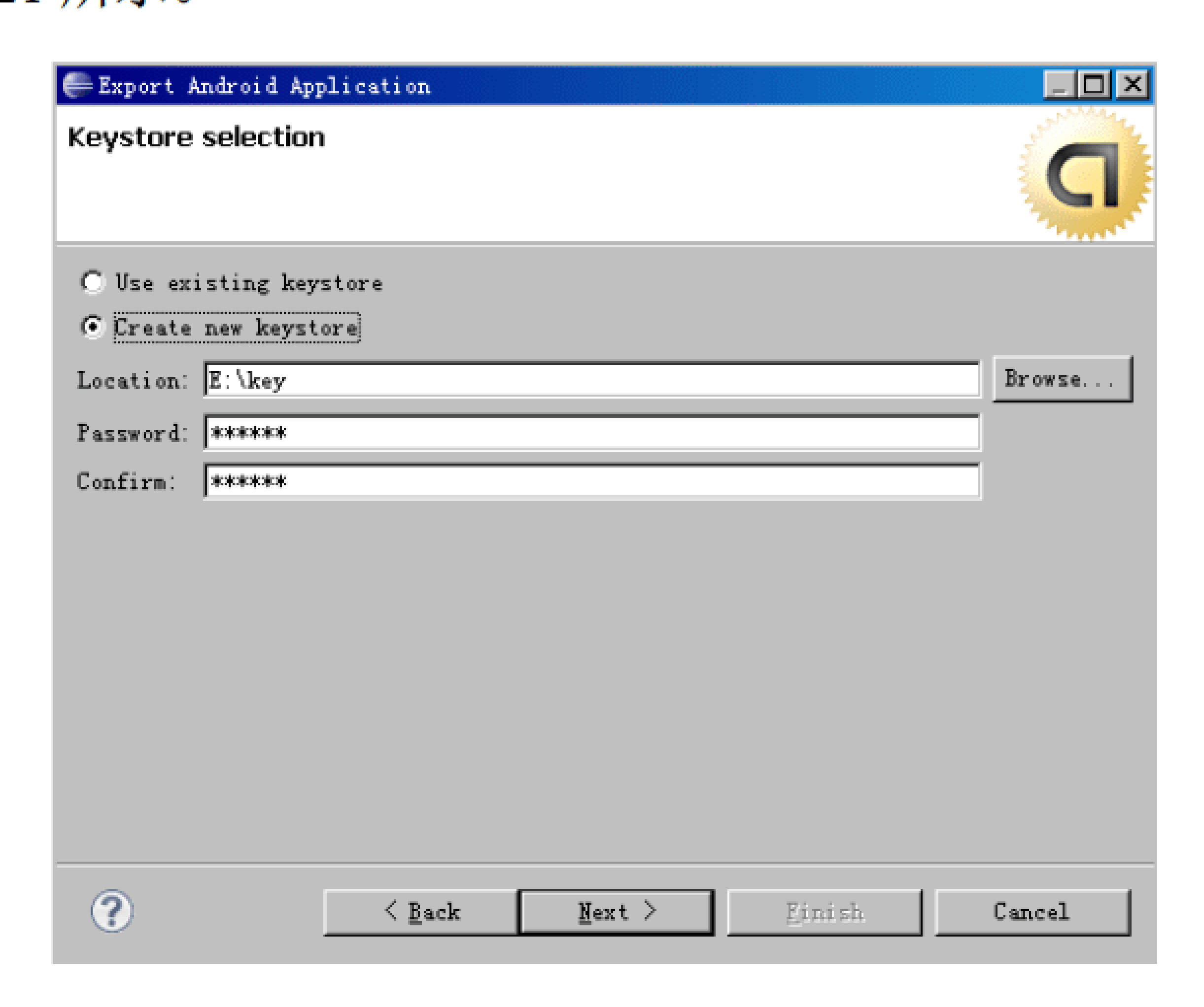


图 11-21 输入文件名和密码

(4) 单击 Next 按钮，在弹出的对话框中输入签名文件路径，如图 11-22 所示。

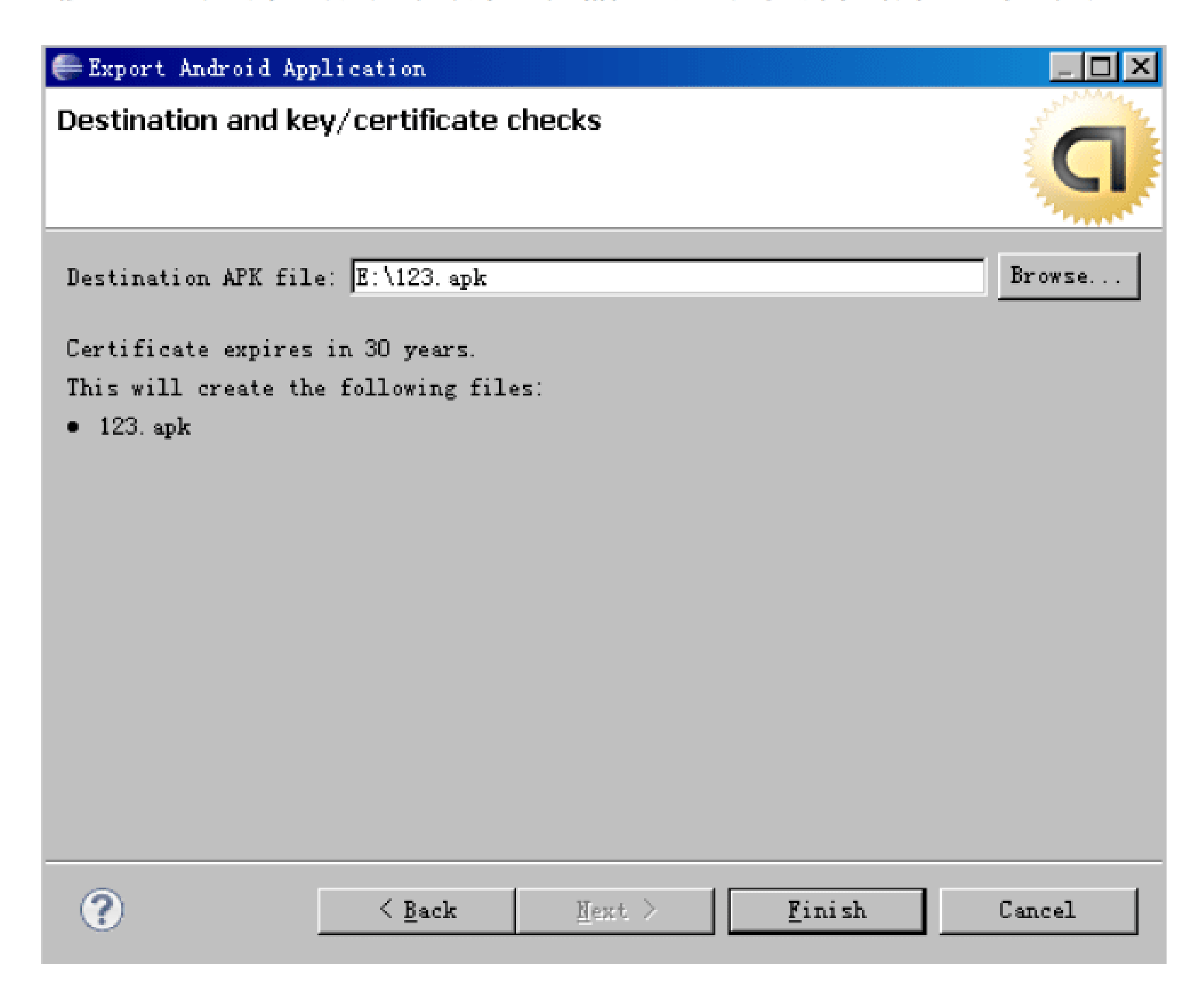


图 11-22　输入信息

(5) 单击 Next 按钮，在弹出的对话框中一次输入签名文件的相关信息，如图 11-23 所示。

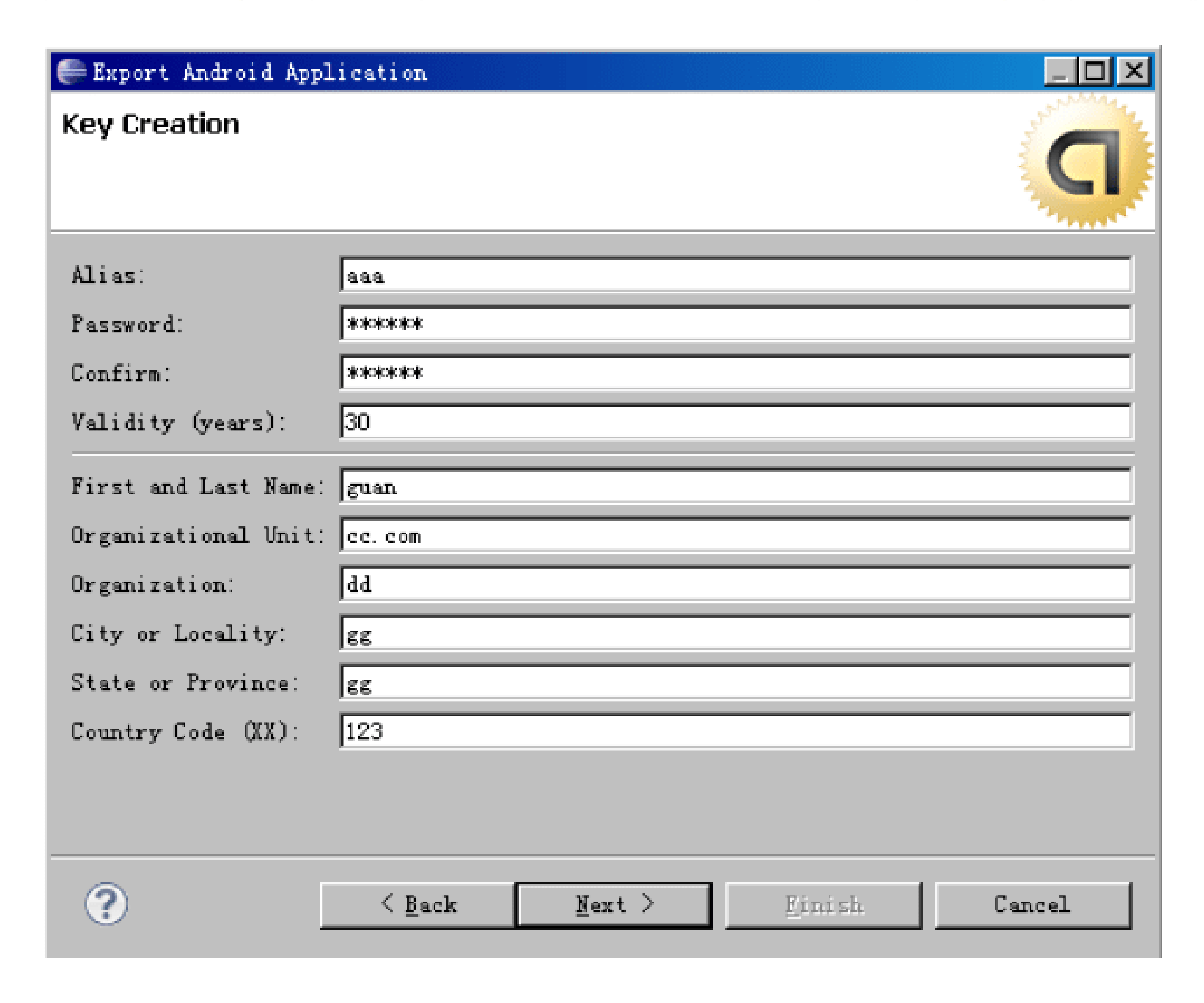


图 11-23　输入信息

(6) 单击 Next 按钮后完成签名文件的创建。

11.3.3　使用签名文件

生成签名文件后，就可以使用它了，在此也有以下两种方式。

1) 命令行

(1) 假设生成的签名文件是 ChangeBackgroundWidget.apk，则最终生成

ChangeBackgroundWidget_signed.apk 为 Android 签名后的 APK 执行文件。

输入以下命令行：

```
jarsigner -verbose -keystore ChangeBackgroundWidget.keystore -signedjar
ChangeBackgroundWidget signed.apk ChangeBackgroundWidget.apk
ChangeBackgroundWidget.keystore
```

上面命令中间不换行。

(2) 按 Enter 键，根据提示输入密钥库的口令短语(即密码)，详细信息如下。

输入密钥库的口令短语：

```
正在添加： META-INF/MANIFEST.MF
正在添加： META-INF/CHANGEBA.SF
正在添加：META-INF/CHANGEBA.RSA
正在签名：res/drawable/icon.png
正在签名：res/drawable/icon_audio.png
正在签名：res/drawable/icon_exit.png
正在签名：res/drawable/icon_folder.png
正在签名：res/drawable/icon_home.png
正在签名：res/drawable/icon_img.png
正在签名：res/drawable/icon_left.png
正在签名：res/drawable/icon mantou.png
正在签名：res/drawable/icon other.png
正在签名：res/drawable/icon pause.png
正在签名：res/drawable/icon play.png
正在签名：res/drawable/icon return.png
正在签名：res/drawable/icon right.png
正在签名：res/drawable/icon_set.png
正在签名：res/drawable/icon_text.png
正在签名：res/drawable/icon_xin.png
正在签名：res/layout/fileitem.xml
正在签名：res/layout/filelist.xml
正在签名：res/layout/main.xml
正在签名：res/layout/widget.xml
正在签名：res/xml/widget info.xml
正在签名：AndroidManifest.xml
正在签名：resources.arsc
正在签名：classes.dex
```

通过上述过程处理后，即可将未签名文件 ChangeBackgroundWidget.apk 签名为 ChangeBackgroundWidget_signed.apk。

在上述方式中，读者可能会遇到以下问题。

问题一：jarsigner 无法打开 Jar 文件 ChangeBackgroundWidget.apk。

解决方法：将要进行签名的 APK 放到对应的文件下，把要签名的 ChangeBackgroundWidget.apk 放到 JDK 的 bin 文件里。

问题二：jarsigner 无法对 jar 进行签名：java.util.zip.ZipException: invalid entry comp ressed size (expected 1598 but got 1622 bytes)

方法一：Android 开发网提示这些问题主要是由于资源文件造成的，对于 Android 开发

来说，应该检查 res 文件夹中的文件，逐个排查。这个问题可以通过升级系统的 JDK 和 JRE 版本来解决。

方法二：这是因为默认给 APK 做了 debug 签名，所以无法做新的签名，这时就必须右击工程，选择 Android Tools | Export Unsigned Application Package 命令，或者从 AndroidManifest.xml 的 Exporting 上也是一样的。然后再基于这个导出的 unsigned apk 做签名，导出的时候最好将其目录选择放在你之前产生 keystore 的那个目录下，这样操作起来就方便了。

2) Eclipse 的 ADT 生成

实际上，使用 Eclipse 可以更加直观、方便地生成签名文件，具体流程如下。

(1) 右击 Eclipse 项目名，依次选择 Android Tools | Export Unsigned Application Package 命令，如图 11-24 所示。

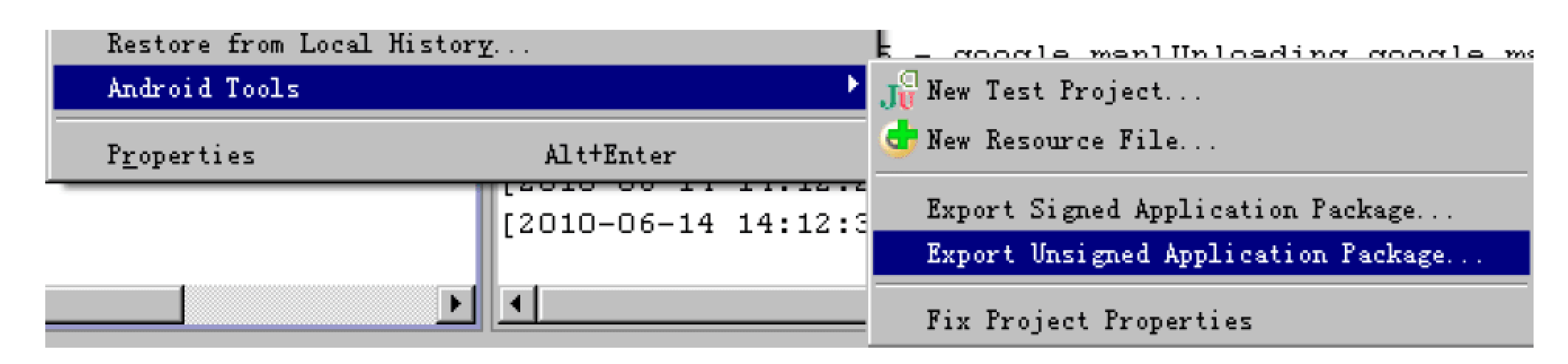


图 11-24　选择 Export Unsigned Application Package 命令

(2) 在弹出的对话框中选择项目，如图 11-25 所示。

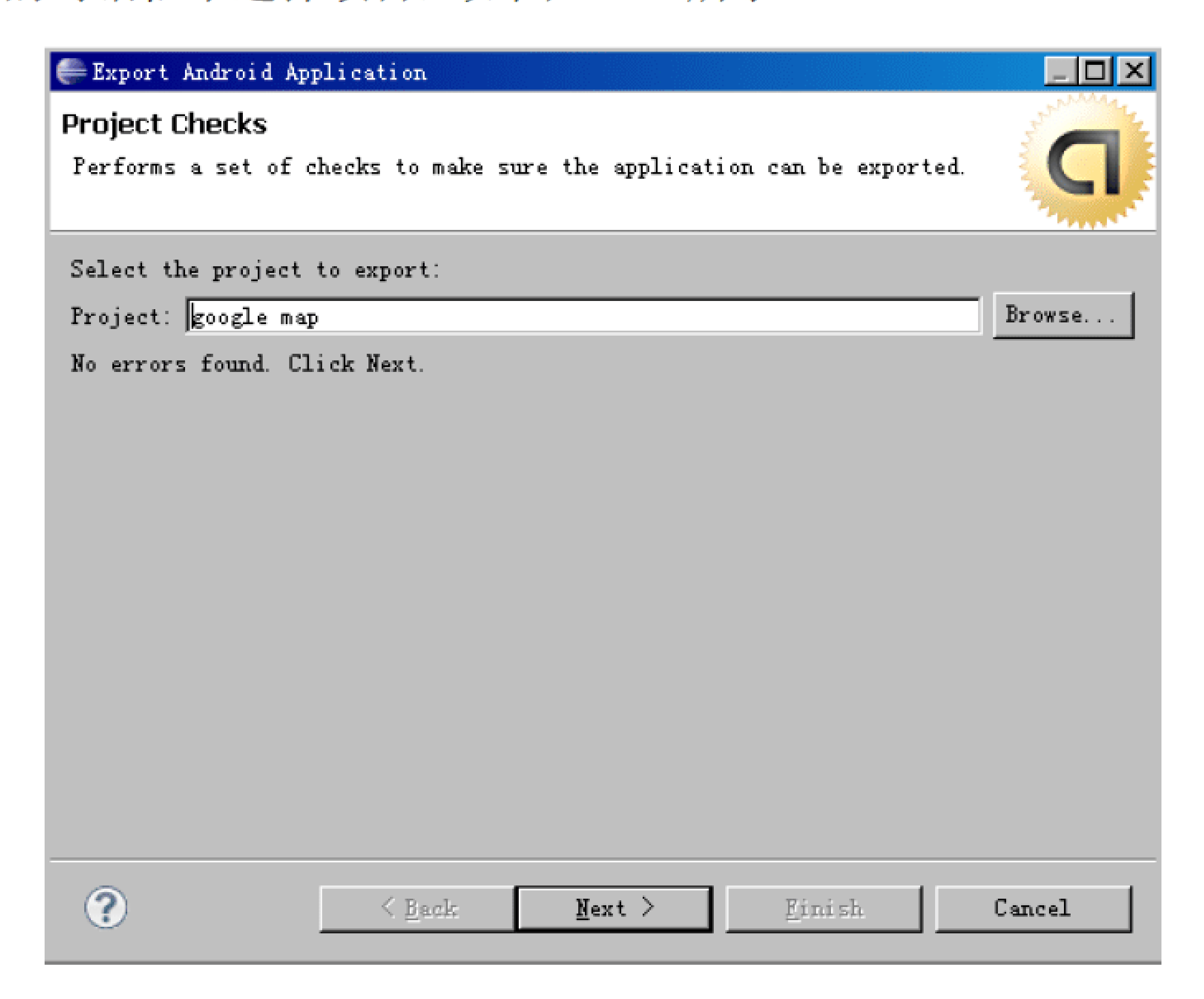


图 11-25　选择项目

(3) 单击 Next 按钮，在弹出的对话框中选中 Use existing keystore 单选按钮，并输入文件的密码，如图 11-26 所示。

(4) 单击 Next 按钮，输入原来签名文件的资料和密码，按照默认提示完成签名。



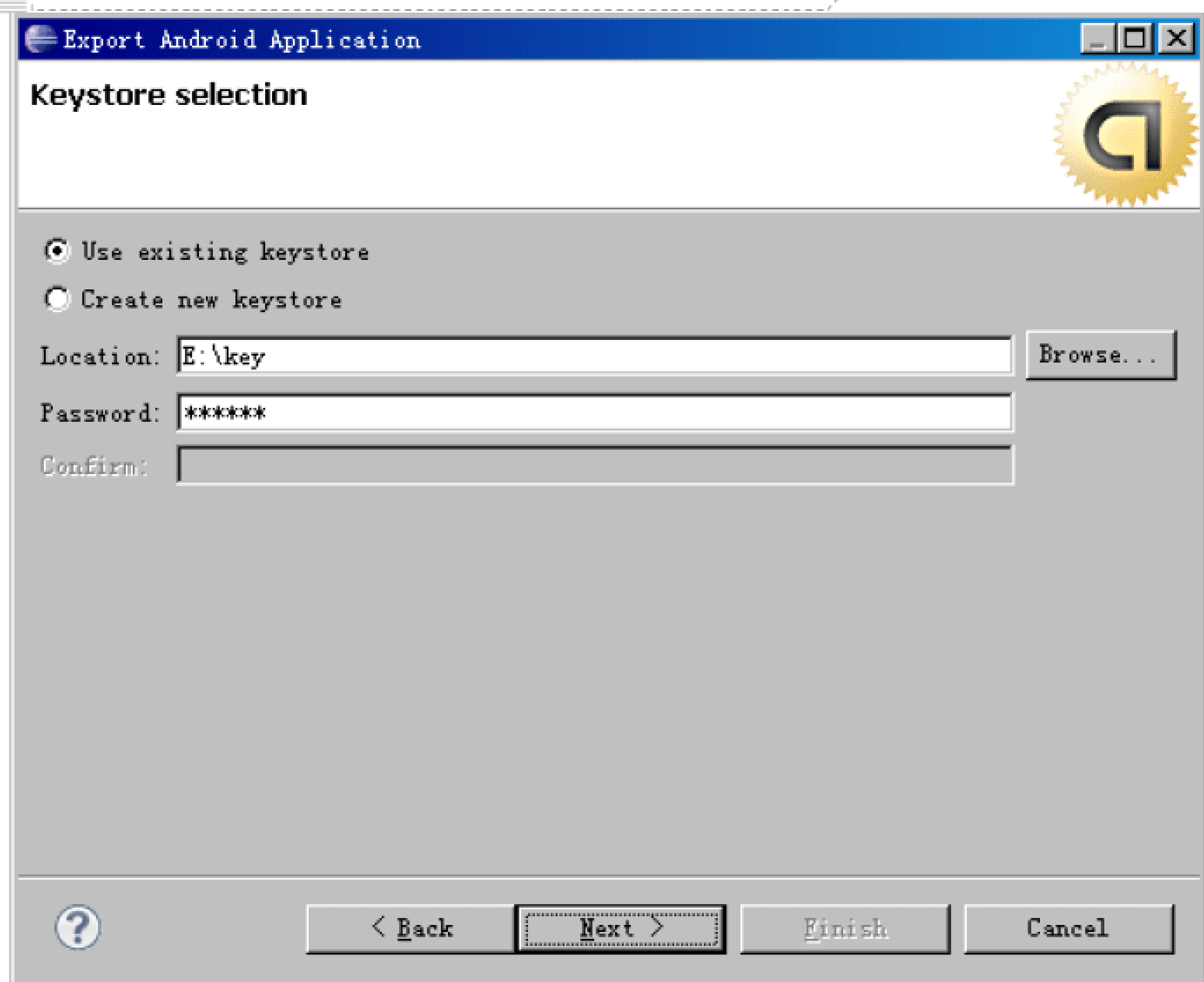


图 11-26　输入密码

11.3.4　发布

发布的过程比较简单，来到 Market，登录个人中心，上传签名后的文件即可。具体操作流程在 Market 站点上有详细说明。为节省本书的篇幅，在此将不做详细介绍。

Android

第12章

综合实例——Android 足球游戏

手机游戏开创了一种全新的娱乐方式和应用模式，并随着移动互联网的发展而火爆。我们在大街上、公交车上、公园中随处可见玩手机游戏的人。所以开发一个 Android 手机游戏很有必要。在本章的内容中，将通过一个具体实例的实现过程，介绍开发一个 Android 游戏的基本流程。本章源码保存在网络资源：daima\12\文件夹中。

12.1 手机游戏产业的发展

手机游戏开创了一种全新的娱乐方式和应用模式，并随着移动互联网的发展而火暴。根据权威部门 itongji 的统计，2011 年第 3 季度中国手机游戏用户数量突破 1.7 亿，环比增长 12.1%，同期手机游戏市场规模达到 12.175 亿。

12.1.1 1.2 亿手机游戏用户

手机游戏异常火暴，运营商们都用实际行动来抢夺这块蛋糕。纷纷投建各自的手机游戏产品基地，为拓宽手机游戏市场做足准备。最近，中国电信游戏运营中心在江苏挂牌成立，“爱游戏”产品同步上线，手机游戏在业内掀起一拨小高潮。

与此同时，国内的手机游戏开发商也不断加大对新款游戏的开发力度，并积极建立更多的非运营商渠道。看他们的实际行动：

(1) 盛大游戏宣布以 8000 万美元的价格全资收购美国游戏分销和内置广告平台 MochiMedia，而 MochiMedia 公司 CEO Jameson Hsu 声称，进入盛大麾下之后将会注重社区游戏和手机游戏上的布局。

(2) 近期宣布将引入手机游戏平台 OpenFeint 的第九城市，7 月初就已宣布与 OpenFeint 的母公司——美国手机游戏及平台公司 Aurora Feint 达成最终投资协议，对其进行战略性少数股权投资。九城还透露有望于年内推出多个手机游戏领域项目。

(3) 几年前已涉足手机游戏业务的腾讯，目前也透露“看好高端手机的游戏平台”，并已成立工作室，专门做高端手机平台的游戏研发和平台研发。英特尔也将向手机游戏平台 OpenFeint 投资 300 万美元。

另外，在终端生产方面，新的手机型号层出不穷，对各种手机游戏的支持也日趋全面化。同时，手机游戏的内容和体验均有较高的提升。

12.1.2 淘金的时代

据 CNNIC 数据显示，截至 2011 年上半年，我国 5.2 亿网民中有 3.57 亿是手机网民，可见我国网民的互联网使用习惯正在日趋显现。正如分析人士所指，移动互联网已成各大互联网巨头重金争夺之地。

目前，移动互联网用户在手机网游、手机阅读、移动微博等细分领域方面的需求表现更为迫切，这使得移动互联网应用服务得以快速发展，移动娱乐等各方面应用表现更为突出。而对于这些用户而言，手机游戏无疑是移动娱乐的先锋应用，市场前景好。

清科研究中心日前发布的报告显示，手机游戏领跑移动互联网行业投资。据悉，2001—2010 年中期，中国移动互联网各细分投资领域中，手机游戏共发生 33 起投资案例，占总投资案例数的 27.97%，已披露金额的投资案例为 26 起，总投资金额为 12045 万美元，平均投资额为 463 万美元。

随着 3G 的普及以及智能手机的推广，移动互联网在我国已呈现成熟迹象，逐渐成为



众厂商觊觎的领域。毫无疑问，智能手机在生活中所扮演的角色越来越重要，甚至将成为手机换代的趋势。而在智能手机丰富的用户软件平台上，手机游戏、流媒体、手机动漫等仍将是推动无线增值业务高增长的主要业务。特别是手机游戏业务，一直是 3G 产业中发展最重要的一个淘金点，受到各厂家的狂热追捧。

有分析人士称，“因为智能手机高质量的触摸屏，强大的程序处理器，优化的图像及摄像功能，更大的内存容量，加速器和 GPS 等功能都变得更加标准，所以更有利于提高手机游戏体验。”随着智能手机的快速普及，尤其是 iPhone、Android 和 iPad 带来的一种热潮，都对手机游戏成为先锋应用有很好的促进作用。

12.1.3　手机游戏的未来发展

虽然整个产业蓬勃发展，但是现实却改变不了。现实是我国手机游戏产业暂处于起步阶段，离“跨越式”发展还存在很大的差距，其进一步发展仍需克服不少难题。首先，短信代收的三次确认影响了手机单机游戏开发商向用户有效收费；其次，手机游戏开发商重视产品数量大于产品质量，而产品从类型和题材上又存在较为严重的同质化现象；再次，虽然手机游戏运营平台呈现多元化的利好趋势，但目前正处在调整期，市场出现观望现象；另外，伴随移动终端和通信网络的发展，将出现更多的手机娱乐方式，如何给用户全新的体验，让用户选择手机游戏，也是摆在眼前的重大难题。

然而，伴随着智能手机、移动网络的不断升级，手机游戏发展空间也越来越大。从目前市场规模和用户规模来看，手机游戏还是一个潜力巨大、尚待发掘的领域，需要在内容、传输渠道、营销、收费等方面加以改进，谋求更广阔的发展出路。

但归根到底，手机游戏产业的发展最重要的还是创意和内容。从目前市场上产品类型来看，手机游戏产品中用户体验还有较大的提升空间，关键在于如何创新产品、改进内容，如何打破千篇一律的游戏风格，跟进移动互联网用户需求的更新换代。

手机游戏产业发展至今，也呈现出了与其他创意产业融合化发展的趋势，产品逐渐向“泛娱乐化”发展。一方面，手机游戏可汲取融合优秀的影视、文学作品等创意产业特色资源，弥补手游行业内容单一的缺陷；另一方面，也将使手机游戏产品多样化，成长为“泛娱乐化”产品，为用户提供更完美的体验。例如近期火热的手机游戏《大玩家》，便是遵照电影剧本、对白设计而成，漫画了群星的剧中造型，不仅有多种视觉效果，还有真实展现的战斗场景。

12.2　Java 游戏开发基础

手机游戏是指运行于手机上的游戏软件。当前开发游戏项目的最常用语言是 Java，例如常见的 J2ME。随着科技的发展，现在手机的功能也越来越多，越来越强大。而手机游戏也随之逐渐发展，早已经不是印象中的诸如俄罗斯方块和贪吃蛇之类画面简陋、规则简单的游戏，进而发展到了可以和掌上游戏机媲美，具有很强的娱乐性和交互性的复杂形态了。现实是买一个好的手机就能够满足你所有路途中的大部分游戏娱乐功能了。一款典型 Java 游戏的开发流程如图 12-1 所示，基本过程的具体说明如下。

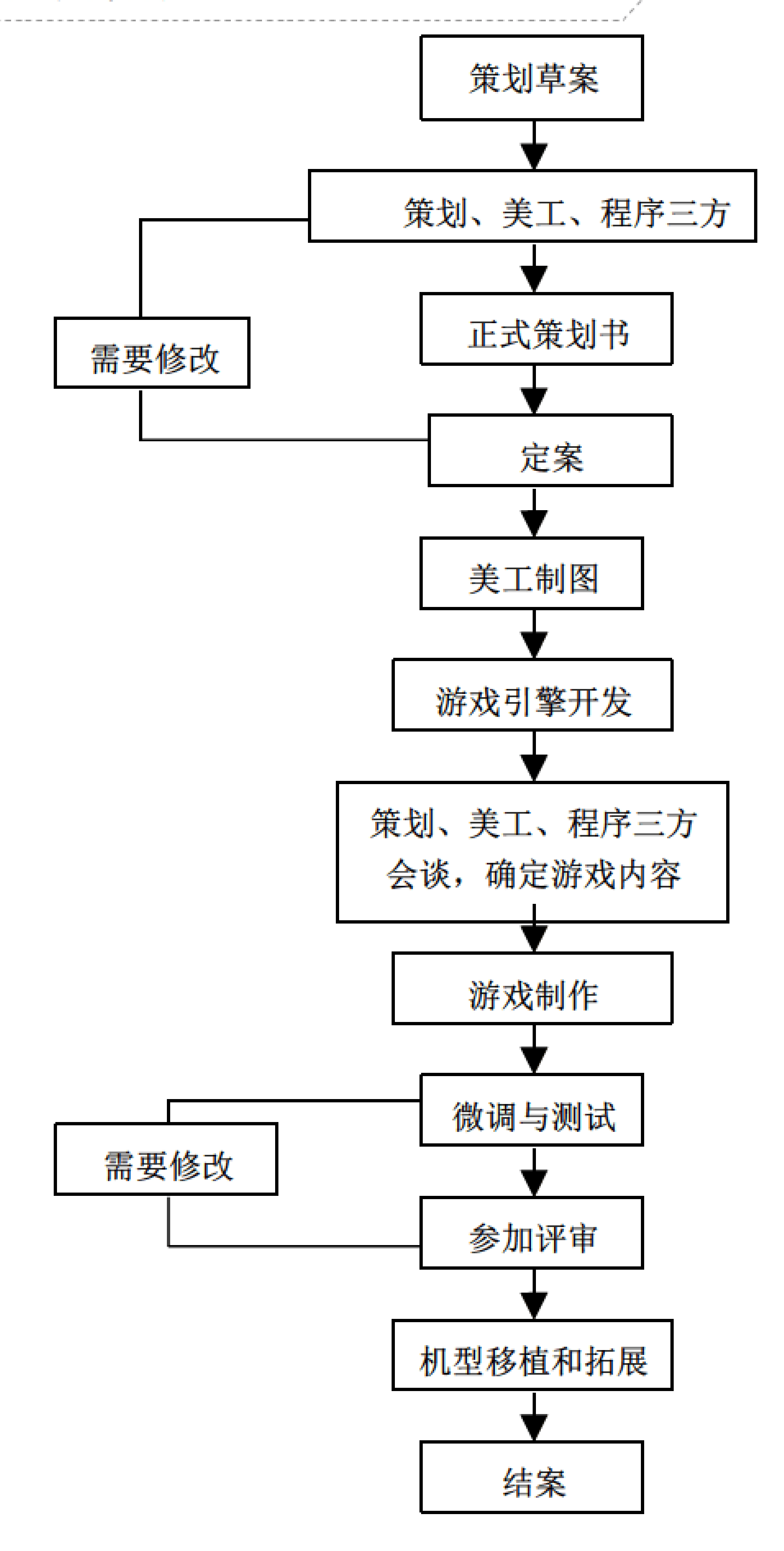


图 12-1　典型 Java 游戏开发流程

1. 立项

在制作游戏之前，策划首先要确定一点：到底想要制作一款什么样的游戏？而要制作一个游戏并不是闭门造车，一个策划说了就算数的简单事情。制作一款游戏要受到多方面的限制：

(1) 市场：即将做的游戏是不是具备市场潜力？在市场上推出以后会不会被大家所接受？是否能够取得良好的市场回报？

(2) 技术：即将做的游戏从程序和美术上是不是完全能够实现？如果不能实现，是不是能够有折中的办法？

(3) 规模：以现有的资源是否能很好地协调并完成即将要做的游戏？是否需要另外增加人员或设备？

(4) 周期：游戏的开发周期是否长短合适？能否在开发结束时正好赶上游戏的销售旺季？

(5) 产品：即将做的游戏在其同类产品中是否有新颖的设计？是否能有吸引玩家的地方？如果在游戏设计上达不到革新，是否能够在美术及程序方面加以补足？如果同类型的游戏市场上已经有了很多，那么即将做的游戏的卖点在哪里？

以上各个问题都是要经过开发组全体成员反复进行讨论才能够确定下来的，需要大家一起集思广益，共同探讨一个可行的方案。如果对上述全部问题都能够有肯定的答案的



话，那么这个项目基本是可行的。但是即便项目获得了通过，在进行过程中也可能会有种种不可预知的因素导致意外情况的发生，所以项目能够成立，只是游戏制作的刚开始。

2. 大纲策划的进行

游戏大纲关系到游戏的整体面貌，当大纲策划案定稿以后，没有特别特殊的情况，是不允许进行更改的。程序和美术工作人员将按照策划所构思的游戏形式来架构整个游戏，因此，在制定策划案时一定要做到慎重和尽量考虑成熟。

3. 正式制作

当游戏大纲策划案完成并讨论通过后，游戏就由三方面开始共同制作。在这一阶段，策划的主要任务是在大纲的基础上对游戏的所有细节进行完善，将游戏大纲逐步填充为完整的游戏策划案。根据不同的游戏种类，所要进行细化的部分也不尽相同。

在正式制作的过程中，策划、程序、美工人员进行及时和经常性的交流，了解工作进展以及是否有难以克服的困难，并且根据现实情况有目的的变更工作计划或设计思想。三方面的配合在游戏正式制作过程中是最重要的。

4. 配音、配乐

在程序和美工进行的差不多要结束的时候，就要进行配音和配乐的工作了。虽然音乐和音效是游戏的重要组成部分，能够起到很好的烘托游戏气氛的作用，但是限于 J2ME 游戏的开发成本和设置的处理能力，这部分已经被弱化到可有可无的地步了。但仍应选择跟游戏风格能很好配合的音乐当作游戏背景音乐，这个工作交给策划比较合适。

5. 检测、调试

游戏刚制作完成，在程序上肯定会有很多错误，严重情况下会导致游戏完全没有办法进行下去。同样，策划的设计也会有不完善的地方，主要在游戏的参数部分。参数部分的不合理，会导致影响游戏的可玩性。此时测试人员需检测程序上的漏洞和通过试玩，调整游戏的各个部分参数使之基本平衡。

12.3　足球游戏介绍

足球游戏是指以足球作为游戏主题的游戏，目前在市场上主根分为足球类小游戏、足球类网页游戏、足球类电视游戏、足球类电脑游戏等几大类别。在足球游戏中，用户可以扮演不同的角色，扮演球员角色的通常以操作类的足球游戏为主，代表作品包括 FIFA 系列、实况足球系列。用户也可以扮演经理人角色，代表作品分别是足球经理人系列游戏。不同类型的足球游戏，可以让玩家得到不同的体验。足球游戏平台包括基于 PSP、PS3、PS2、NDSL、PC、手机、XBOX360 等，例如屏幕视图、道具视图、角色视图等。

12.3.1　手机足球游戏

绿茵场是充满激情的地方，当今足球是世界第一普及性运动，每 4 年一届的世界杯堪

比奥运会。足球运动的盛行，也衍生了很多附属产业的兴起，例如手机游戏和电脑游戏。电脑足球游戏相比大家不会陌生，例如著名的 EA FIFA 和实况足球曾经令我们陶醉其中。现在随着智能手机的蓬勃发展，手机足球游戏也日益受到人们的青睐。

本项目基于 Android 平台，开发一个基本的足球游戏。操作方式比较简单，就像桌式足球一样用球杆来控制多个运动员，使玩家在手机中体验运动带来的刺激和魅力。

12.3.2 策划游戏

本项目属于体育类游戏，下面开始策划整个项目的具体功能。

(1) 情节

作为一个竞技足球项目，需要模拟现实世界的足球实况，所以游戏情节都是几乎一样的。在此阶段的主要工作是，规划游戏进程和规划不同的场景。

(2) 目标用户

本项目的玩家主要是对足球有一定了解的用户，或者是对体育运动特别是足球感兴趣的人，并且以年轻人为主。

(3) 运行平台

本项目的运行平台是 Android 4.0。

(4) 显示技术

为了将绿荫场景生动的展示在用户面前，需要采用 2D 单屏模式以指定的视角展示游戏。

(5) 操控方式

将使用手机键来控制本游戏。

12.3.3 准备工作

在进行游戏开发之前，需要准备好游戏中用到的图片素材和配音文件。其中用到的图片素材文件保存在 res\drawable-mdpi 目录下，如图 12-2 所示。

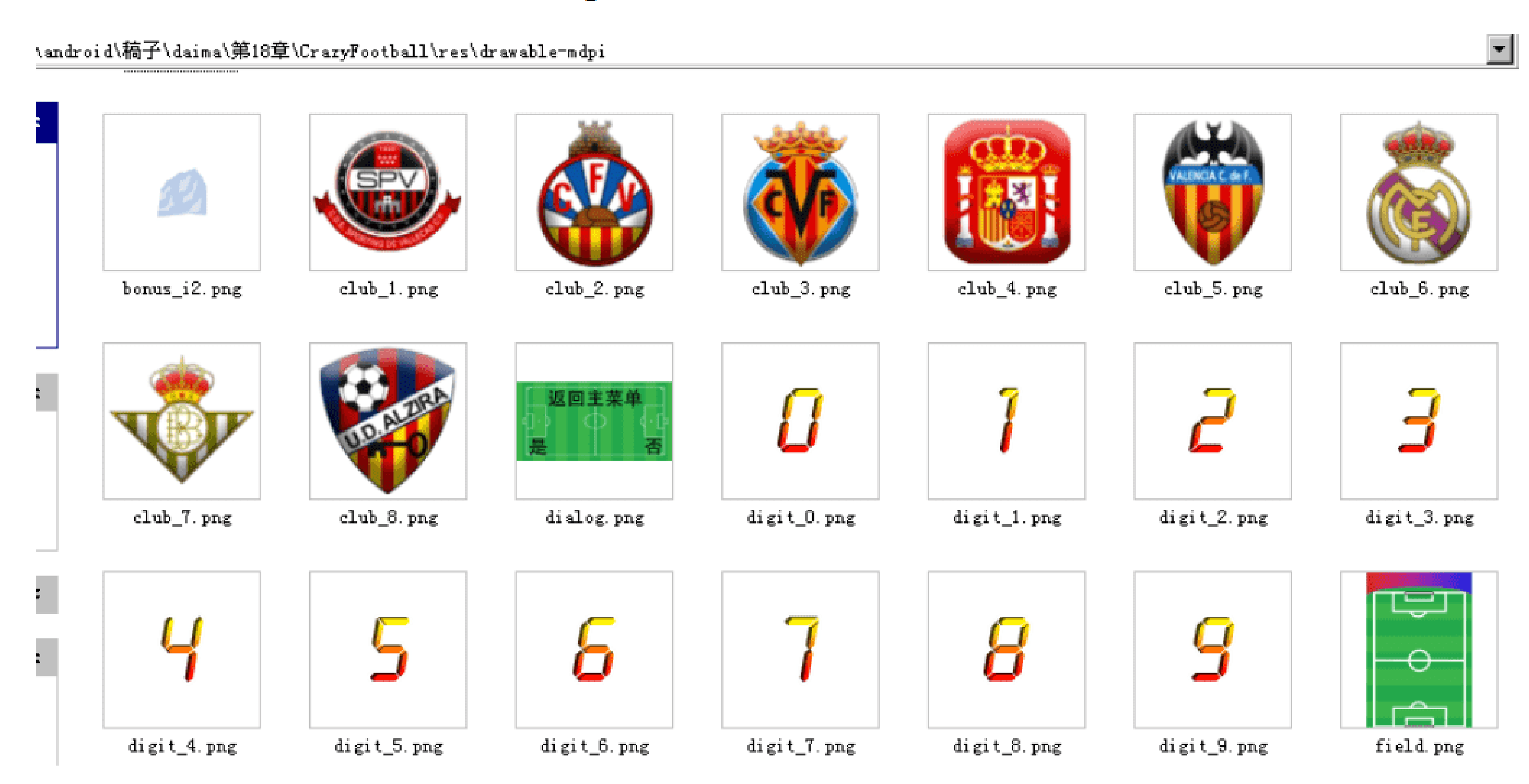


图 12-2　图片素材

用到的配音文件保存在 res\raw 目录下，如图 12-3 所示。

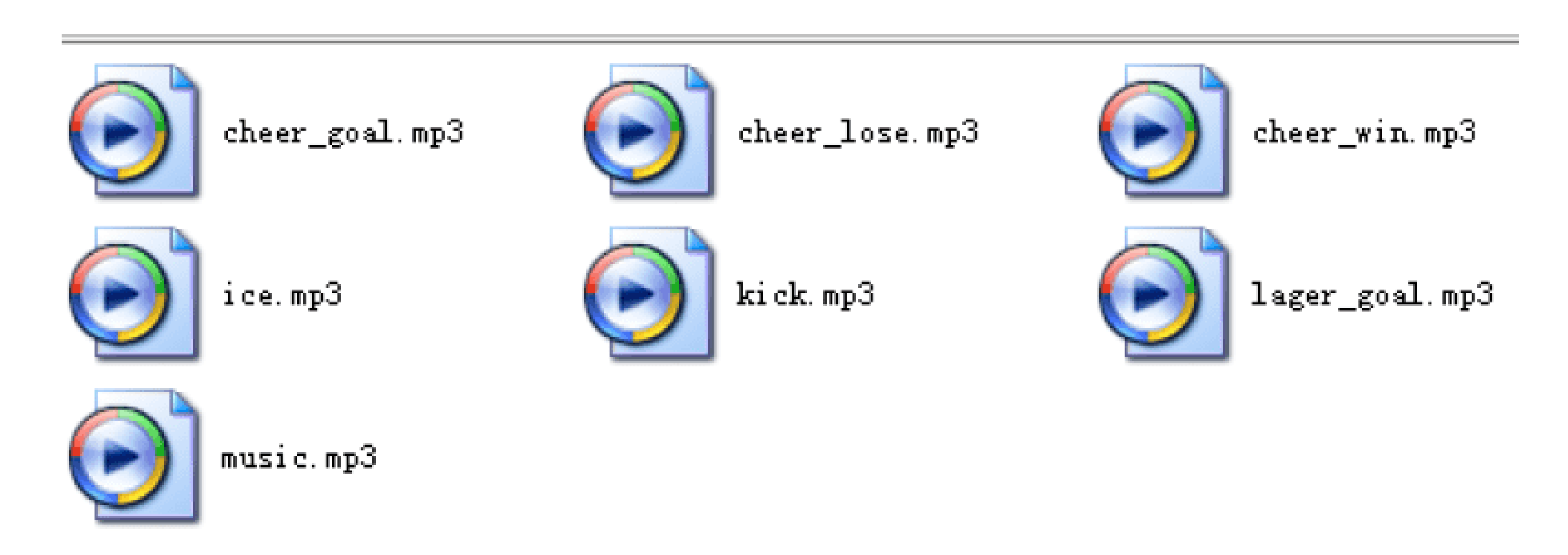


图 12-3　配音文件

12.4　项 目 架 构

在本节内容中，将对整个项目进行总体架构分析，并对项目中的各个类及其结构进行一一介绍。

12.4.1　总体架构

本项目的总体架构如图 12-4 所示。

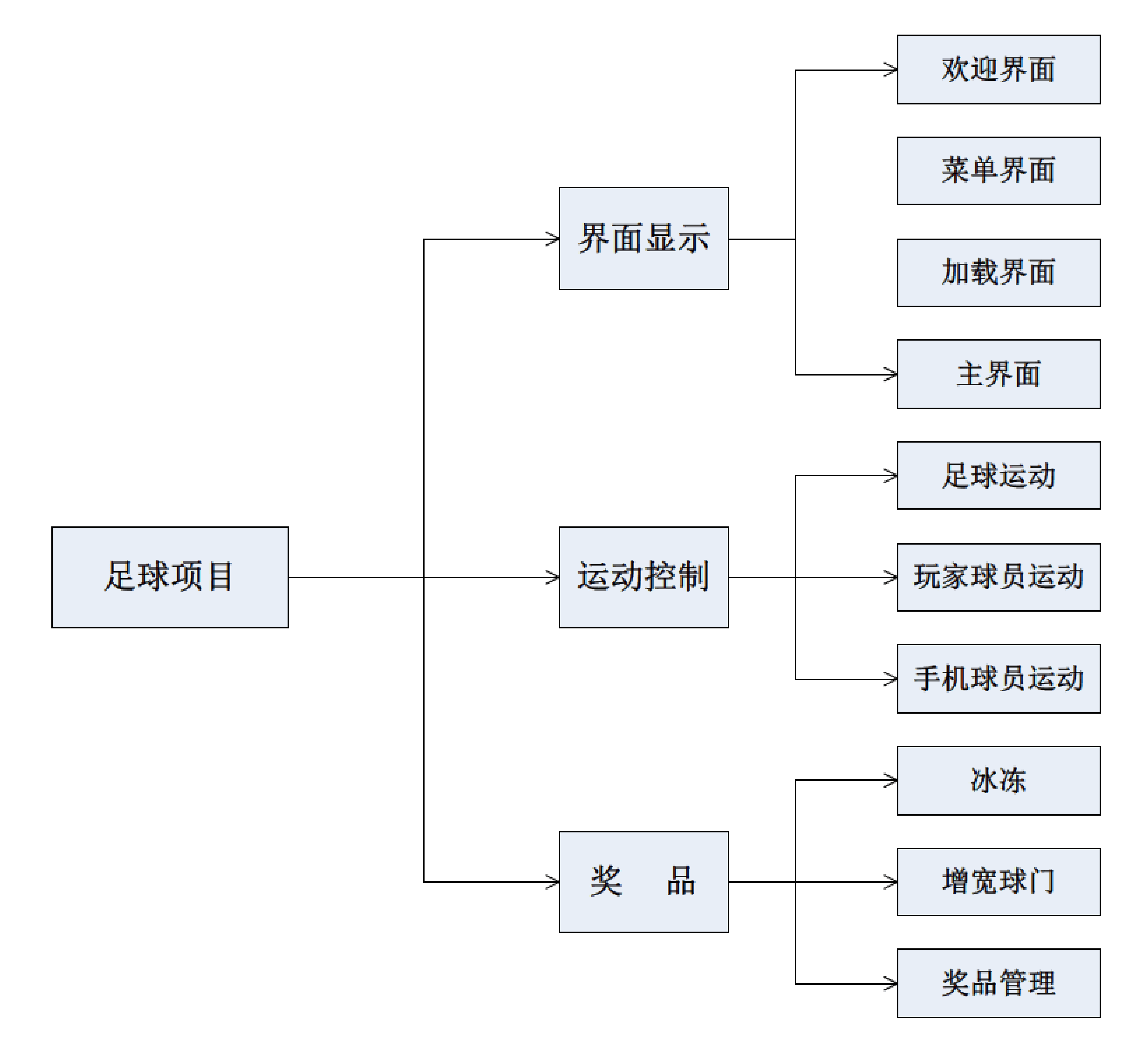


图 12-4　总体架构图

12.4.2 规划类

类是面向对象的核心，在本游戏项目中，为了实现各个具体的功能，需要编写各个类来实现具体的功能。下面将介绍各个类的具体功能。

1. 界面显示类

项目中和界面显示有关的类如下。

(1) 类 FootballActivity：继承自 Activity，扮演一个类似于控制器的角色，能够在不同的视图之间切屏及处理键盘和触控笔单击事件。

(2) 视图类：包含了几个继承与 SurfaceView 的视图类，有 WelcomeView、LoadingView 和 GameView，能够为玩家显示不同的视图效果。各类的具体功能如下：

- WelcomeView：显示欢迎界面和系统菜单；
- LoadingView：显示在不同界面之间进行切换时的进度条；
- GameView：显示游戏画面。

(3) 线程类：包含了几个继承与 Thread 的线程类，能够实现刷屏和修改后台数据库的功能。各类的具体功能如下：

- WelcomeDrawThread、LoadingDrawThread 和 DrawThread：能够刷新视图类中的显示内容；
- WelcomeThread：能够修改 WelcomeView 中的数据。

(4) 类 CustomGallery：这是一个自定义控件，能够实现类似于 Gallery 的图片显示效果。

2. 运动控制类

项目中和运动控制有关的类如下。

(1) Ball 类：根据足球的方向移动足球的位置，并检测是否与双方球员或奖励物品发生碰撞，如果碰撞则进行碰撞处理。

(2) AIThread 类：使用算法设置手机控制球员的移动方向，项目中的球员只有左移和右移两种方向。

(3) PlayerMoveThread 类：定时读取球员移动方向，根据方向来移动球员。

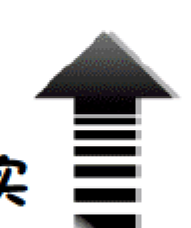
(4) Play 类：封装了球员信息以及对这些信息进行操作的成员方法。

3. 奖品类

在奖品类中，有继承自 Object 类的 Bonus 类，此类是奖品类的父类。还有继承自 Thread 类的 BonusManager 类，此类能够添加一些新的 Bonus 对象到游戏中。

12.5 Android 手机游戏的优化策略

因为游戏开发只是重要的 Android 应用开发之一，所以接下来将要讲解优化策略，其实已经在本书前面的内容中讲解过了。总结来说，Android 手机游戏的优化策略主要包括如下三个部分。



(1) 优化绘图

- 脏矩形：每次都重绘整个背景图，其实是非常浪费的，前后两帧的图其实只有很少的一部分发生了变化，因此可以只重绘变化的部分。这是一种常用的绘图优化方式，需要注意的是，android 用了双缓冲，也就是说，使用脏矩形的时候，需要连续绘制两次才能完成对 surface 的刷新。
- 卷屏：这是常用的一种方法，缓存整张背景图，抽象出一个可视窗口，仅显示窗口中的内容，窗口的移动方向与 sprite 相同，与背景运动方向相反。Android 游戏背景图的分辨率一般与屏幕的相同，这种方法很少会被用到。

(2) 优化引擎

- 流水作业化资源：简单来说，就是整合资源，不用的资源就及时释放，需要用到的资源再加载，类似流水线生产过程。比如，游戏加载过程中，当前关卡(场景、模式等)使用不到的音乐或者图像资源就全部释放，仅加载需要用的资源；用不到的线程，不要让它休眠，一定要把它干掉；如果有的资源只用得到一部分，那么就拆解开来，仅加载需要的部分。
- 状态转移逻辑：游戏开发前一定要想清楚状态转移，冗余的状态变化将损耗框架的整体性能，对游戏流畅性的影响以及后期修改的成本往往是远远超出预期的。不要怕费力，一定要认真优化状态转移过程。此外，Activity 之间切换、UI 线程和游戏线程之间的切换，都是非常花费时间的，应该尽力避免。

(3) 逻辑优化

- 预处理：尽可能地预处理游戏逻辑中的运算。比如游戏中经常要用到随机数，就应该在游戏开始之前，生成足够的随机数供游戏逻辑调用，千万避免使用系统自身的 rand()函数。这种优化方式难度比较大，但是往往是突破瓶颈的最有效手段。
- 算法优化：这个没有什么好多说的，算法功底和经验积累很重要，单干是搞不定的，赶快找同事帮忙。
- 语法优化：语法对运行速度也有很大影响，比如 for 循环，不同的写法，时间开销差别极大。

12.6 具体编码

经过前面的讲解，整个项目的前期工作结束。从本节的内容开始，将进入具体编码阶段。希望读者仔细品味本节的内容，真正步入 Android 开发高手的殿堂。

12.6.1 Activity 类开发

在 Android 中，Activity 负责不同界面间的切换。在本项目中，Activity 还能够实现按键单击和修改按键状态的功能。本足球游戏项目的 Activity 类是由文件 FootballActivity.java 实现的，下面开始介绍它的实现流程。

(1) 作为一个控制器，首先要声明项目中需要的成员变量，在代码中对这些成员的具体含义进行了详细说明注释。具体代码如下。

```
package wyf.wpf;                        //声明包语句
import android.app.Activity;            //引入相关类
import android.content.Context;
import android.graphics.Rect;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.os.Looper;
import android.view.KeyEvent;
import android.view.MotionEvent;
import android.view.View;
import android.view.Window;
import android.view.WindowManager;
/*
 * 游戏的主类，负责切换视图，接收和捕获用户的键盘输入并做相应处理。
 * 游戏的欢迎 View，加载进度的 View 和游戏视图 View 在这里都有引用，可以
 * 切换，通过 onTouchEvent 方法处理函数来接受用户点击屏幕事件
 */
public class FootballActivity extends Activity{
    View current;                       //记录当前 View
    GameView gv;                        //GameView 对象
    WelcomeView welcome;                //欢迎界面
    LoadingView lv;                     //进度条加载界面

    int keyState = 0;                   //xxxx00 为不动，xxxx10 为向左,xxxx01 为向右
    PlayerMoveThread pmt;               //移动球员位置的线程
    boolean wantSound = true;           //是否播放声音标志位
    int [] layoutArray;                 //表示球员球场站位的数组
    MediaPlayer mpWelcomeMusic;         //游戏开始前的欢迎音乐
    MediaPlayer mpKick;                 //踢球音效
    MediaPlayer mpCheerForWin;          //赢了的音乐
    MediaPlayer mpCheerForLose;         //输了的音乐
    MediaPlayer mpCheerForGoal;         //进球后的音乐
    MediaPlayer mpIce;                  //撞到冰山后的音乐
    MediaPlayer mpLargerGoal;           //撞到打开球门后的音乐
    Rect [] rectPlus;                   //代表增加球员按钮的矩形框
    Rect [] rectMinus;                  //代表减少球员按钮的矩形框
    Rect rectSound;                     //是否播放声音按钮的矩形框
    Rect rectStart;                     //开始按钮的矩形框
    Rect rectQuit;                      //退出按钮的矩形框
    Rect rectGallery;                   //表示 Gallery 的矩形框
    int [] imageIDs ={                  //存放 8 个俱乐部的图片 ID
            R.drawable.club 1,
            R.drawable.club 2,
            R.drawable.club_3,
            R.drawable.club_4,
            R.drawable.club 5,
            R.drawable.club 6,
            R.drawable.club 7,
            R.drawable.club 8
        };
    int clubID = imageIDs[0];           //记录用户选择的俱乐部的 ID
```

(2) 编写 onCreate 重写方法，这个方法在 Activity 创建时被首先调用，能够对各个变量进行初始化处理。具体代码如下。

```
@Override
   public void onCreate(Bundle savedInstanceState) {          //重写
onCreate 方法
      super.onCreate(savedInstanceState);
      initWelcomeSound(this);      //初始化声音库
      requestWindowFeature(Window.FEATURE_NO_TITLE);          //设置全屏
      getWindow().setFlags(
          WindowManager.LayoutParams.FLAG_FULLSCREEN,
          WindowManager.LayoutParams.FLAG_FULLSCREEN
          );
      welcome = new WelcomeView(this);           //将屏幕切到欢迎界面
      setContentView(welcome);
      current = welcome;
      if(wantSound && mpWelcomeMusic!=null){          //如需要，播放相应声音
       mpWelcomeMusic.start();
      }
      initRects();     //初始化用于匹配点击事件的矩形框
   }
```

(3) 定义方法 initWelcomeSound(Context context)和 initRects()，分别初始化欢迎界面的声音和初始化矩形框。具体代码如下。

```
//方法：初始化欢迎界面的声音
   public void initWelcomeSound(Context context){
       mpWelcomeMusic = MediaPlayer.create(context, R.raw.music);
   }
   //方法：初始化矩形框
   public void initRects(){
    rectPlus = new Rect[3];
    rectMinus = new Rect[3];
    for(int i=0;i<3;i++){
        rectPlus[i] = new Rect(244,200+40*i,280,236+40*i);
        rectMinus[i] = new Rect(280,200+40*i,316,236+40*i);
    }
    rectSound = new Rect(135,370,185,420);
    rectStart = new Rect(205,425,295,475);
    rectQuit = new Rect(25,425,115,475);
    rectGallery = new Rect(10,10,310,110);
   }
```

(4) 定义方法 onTouchEvent(MotionEvent event)，能够处理用户所有的屏幕单击事件。具体代码如下。

```
    public boolean onTouchEvent(MotionEvent event) {//重写 onTouchEvent 方法
        if(event.getAction()== MotionEvent.ACTION_UP){//判断事件类型
            int x = (int)event.getX();      //获得点击处的 X 坐标
            int y = (int)event.getY();      //获得点击处的 Y 坐标
```




```
            if(current == welcome){//如果当前界面是欢迎界面
                if(rectGallery.contains(x, y)){     //用户点击的是 Gallery
                    welcome.cg.galleryTouchEvnet(x, y);//交给 Gallery 来处理
点击事件
                }
                else if(rectSound.contains(x, y)){      //点下的是声音选项
                    this.wantSound = !this.wantSound;   //更改声音选项
                    return true;
                }
                else if(rectStart.contains(x, y)){      //点下开始键
                    if(checkLayout(welcome.layout)){    //检查玩家选择的布局
是否正确
                        layoutArray = welcome.layout;   //获得玩家选择站位布局
                        lv = new LoadingView(this);     //创建读取进度 View
                        this.setContentView(lv); //将屏幕设为读取进度的 LoadingView
                        this.current = lv;              //记录当前 View
                        lv.lt.start();          //启动 LoadingView 的刷屏线程
                        new Thread(){   //启动一个新线程，在其中创建 GameView 对象
                            public void run(){
                                Looper.prepare();
                                if(wantSound){
                                    initSound();//初始化声音
                                }
                                //创建游戏界面
gv = new GameView(FootballActivity.this,imageIDs[welcome.cg.currIndex]);

                                lv.progress = 100;
                                welcome = null;     //释放掉 WelcomeView
                            }
                        }.start();
                    }
                }
                else if(rectQuit.contains(x,y)){        //按下退出键
                    System.exit(0);                     //程序退出
                }
                else{           //检查是否按下了修改队员站位的加号和减号按钮
            for(int i=0;i<3;i++){
            if(rectPlus[i].contains(x,y)){ //如果有加号按钮点下，就增加对应进
攻防守线上人数
                            //如果有富余的人再加

    if(welcome.layout[0]+welcome.layout[1]+welcome.layout[2] <10){
                                welcome.layout[i]++;
                            }
                            break;
                        }
                    if(rectMinus[i].contains(x, y)){//如果有减号按钮点下，就
减少相应人数
                            if(welcome.layout[i] > 0){  //如果该处人数不为零，
就减少一个
                                welcome.layout[i]--;
```

```
                    }
                    break;
                }
            }
        }
    }
    else if(current == gv){          //如果当前显示的 View 为 GameView
        if(gv.rectMenu.contains(x,y)){ //如果点下了菜单按钮
            gv.isShowDialog = true;    //设置显示对话框
            gv.ball.isPlaying = false; //足球停止移动
            pmt.flag = false;          //使 PlayerMoveThread 空转
        }
    else if(gv.rectYesToDialog.contains(x,y)){//如果点下的是对话框中
的【是】按钮
    if(gv.isShowDialog){               //检查对话框是不是正在显示
          welcome = new WelcomeView(this); //新建一个 WelcomeView
              setContentView(welcome); //设置当前屏幕为 WelcomeView
              welcome.status = 3;          //直接设为待命状态
              current = welcome;               //记录当前屏幕
              gv = null;           //将 GameView 指向的对象声明为垃圾
              if(wantSound && mpWelcomeMusic!=null){
            //如需要，播放声音
               mpWelcomeMusic.start();
              }
            }
        }
else if(gv.rectNoToDialog.contains(x,y)){   //如果点下的是对话框中的【否】按钮
            if(gv.isShowDialog){             //检查对话框是不是正在显示
                gv.isShowDialog = false;       //不显示对话框
                pmt.flag = true;               //设置双方球员可移动
                gv.ball.isPlaying = true;          //设置足球可移动
            }
        }
    }
    else if(current == lv){            //如果当前屏幕为 LoadingView
        if(lv.progress == 100){            //如果进度达到 100%
            setContentView(gv);            //屏幕切换到 GameView
            current = gv;                  //记录当前 View
            lv = null;                     //lv 指向的对象声明为垃圾
            if(mpWelcomeMusic.isPlaying()){     //如需要，播放相应声音
                mpWelcomeMusic.stop();
            }
            gv.startGame();                          //开始游戏
        }
    }
}
return true;
}
```

(5) 定义方法 initSound()，用于加载游戏中用到的声音。具体代码如下。

```
//方法：加载游戏中用到的声音
    public void initSound(){
        mpKick = MediaPlayer.create(this, R.raw.kick);
        updateProgressView();//更新进度条
        mpCheerForWin = MediaPlayer.create(this, R.raw.cheer win);
        updateProgressView();//更新进度条
        mpCheerForLose = MediaPlayer.create(this, R.raw.cheer lose);
        updateProgressView();//更新进度条
        mpCheerForGoal = MediaPlayer.create(this, R.raw.cheer goal);
        updateProgressView();//更新进度条
        mpLargerGoal = MediaPlayer.create(this, R.raw.lager goal);
        updateProgressView();//更新进度条
        mpIce = MediaPlayer.create(this, R.raw.ice);
        updateProgressView();//更新进度条
    }
```

(6) 定义方法 updateProgressView()，用于更新进度条的进度。具体代码如下。

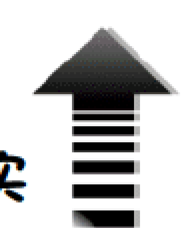
```
//更新进度条的进度
    public void updateProgressView(){
     lv.progress+=15;
    }
     @Override
```

(7) 定义方法 checkLayout(int [] layout)，用于检查用户输入的 layout 是否合法。具体代码如下。

```
//检查用户输入的 layout 合不合法
    public boolean checkLayout(int [] layout){
        int sum=0;
        for(int i=0;i<layout.length;i++){   //遍历存放球员站位的数组
            if(layout[i]<0){         //如果发现某个进攻/防守阵线上的球员为负数
                return false;
            }
            else{
                sum+=layout[i];      //将各个阵线上的球员个数相加
            }
        }
        if(sum == 10){                      //如果和为 10，则该站位合法
            return true;
        }
        else{
            return false;            //返回 false
        }
    }
}
```

12.6.2 欢迎界面

在欢迎界面中，涉及的类有 WelcomeView、WelcomeThread、WelcomeDrawThread 和 CustomGallery。在下面的内容中，将对实现上述类的文件进行一一讲解。



(1) 文件 CustomGallery.java

此文件仿照 Gallery 控件实现了图片的显示，在项目执行之后供用户选择自己的球队。其具体实现代码如下。

```
/*
 * 该类为自定义的 gallery，为实现 Gallery 的效果
 */
public class CustomGallery{
    Bitmap [] bmpContent;       //Gallery 要显示的内容图片
    int length;                 //Gallery 要显示的图片数组大小
    int currIndex;              //当前被显示的图片的索引
    int startX;                 //绘制 Gallery 时其左上角在屏幕中的 X 坐标
    int startY;                 //绘制 Gallery 时其左上角在屏幕中的 Y 坐标
    int cellWidth;              //每个图片的宽度
    int cellHeight;             //每个图片的高度
    //构造器，初始化主要成员变量
    public CustomGallery(int startX,int startY,int cellWidth,int
cellHeight){
        this.startX = startX;
        this.startY = startY;
        this.cellWidth = cellWidth;
        this.cellHeight = cellHeight;
    }
    public void setContent(Bitmap [] bmpContent){  //方法：为 Gallery 设置显
示内容
        this.bmpContent = bmpContent;
        this.length = bmpContent.length;
    }
    public void setCurrent(int index){              //方法：设置当前显示的图片
        if(index >=0 && index < length){
            this.currIndex = index;
        }
    }
    public void drawGallery(Canvas canvas,Paint paint){//方法：绘制自己
        //创建背景的画笔
        Paint paintBack = new Paint();
        paintBack.setARGB(220, 99, 99, 99);
        //创建边框的画笔
        Paint paintBorder = new Paint();
        paintBorder.setStyle(Paint.Style.STROKE);
        paintBorder.setStrokeWidth(4.5f);
        paintBorder.setARGB(255, 150, 150, 150);
        //画左边的图片
        if(currIndex >0){
            canvas.drawRect(startX, startY, startX+cellWidth,
startY+cellHeight, paintBack); //背景
            canvas.drawBitmap(bmpContent[currIndex-1], startX, startY,
paint);                //贴图片
            canvas.drawRect(startX, startY, startX+cellWidth,
startY+cellHeight, paintBorder);   //画左边图片的边框
```

```
        }
        //画被选中的图片
        canvas.drawRect(startX+cellWidth, startY, startX+cellWidth*2,
startY+cellHeight, paintBack);          //背景
        canvas.drawBitmap(bmpContent[currIndex], startX+cellWidth,
startY, paint);                         //贴图片
        //画右边的图片
        if(currIndex<length-1){
            canvas.drawRect(startX+cellWidth*2, startY,
startX+cellWidth*3, startY+cellHeight, paintBack); //背景
           canvas.drawBitmap(bmpContent[currIndex+1], startX+cellWidth*2,
startY, paint);                     //贴图片
            paintBorder.setARGB(255, 150, 150, 150);
                                    //画右边图片的边框
            canvas.drawRect(startX+cellWidth*2, startY,
startX+cellWidth*3, startY+cellHeight, paintBorder);
        }
        //画选中的边框
        paintBorder.setColor(Color.RED);
        canvas.drawRect(startX+cellWidth, startY, startX+cellWidth*2,
startY+cellHeight, paintBorder);
    }
    public void galleryTouchEvnet(int x,int y){        //方法：Gallery 的处
理点击事件方法
        if(x>startX && x<startX+cellWidth){     //点在了左边那张图片
            if(currIndex > 0){                  //判断当前图片的左边还有没有图片
                currIndex --;                   //设置当前图片为左边的图片
            }
        }
        else if(x>startX+cellWidth*2 && x<startX+cellWidth*3){     //点在
了右边那张图片
            if(currIndex < length-1){           //判断当前图片的右边还有没有图片
                currIndex++;                    //设置当前图片为右边的图片
            }
        }
    }
}
    @Override
   public void surfaceChanged(SurfaceHolder holder, int format, int width,
            int height) {                       //重写 surfaceChanged 方法
    }
    @Override
    public void surfaceCreated(SurfaceHolder holder) {     //重写
surfaceCreated 方法
        if(!wt.isAlive()){          //启动后台修改数据线程
            wt.start();
        }
        if(!wdt.isAlive()){         //启动后台绘制线程
            wdt.start();
        }
    }
```

```
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {   //重写
surfaceDestroyed 方法
        if(wt.isAlive()){                //停止后台修改数据线程
            wt.isWelcoming = false;
        }
        if(wdt.isAlive()){               //停止后台绘制线程
            wdt.flag = false;
        }
    }
}
```

在上述代码中，函数 CustomGallery 能够初始化屏幕上的坐标和所显示的图片大小。

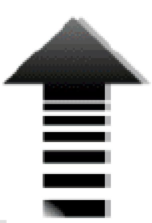
(2) 文件 WelcomeView.java

类 WelcomeView 继承与 SurfaceView 类，并且实现了 SurfaceHolder.Callback 的接口。具体实现代码如下。

```
/*
 * 该类继承自 View，实现欢迎动画的播放，以及主菜单的显示
 */
public class WelcomeView extends SurfaceView implements
SurfaceHolder.Callback{
    WelcomeThread wt;           //后台修改数据线程
    WelcomeDrawThread wdt;      //后台重绘线程
    FootballActivity father;    //Activity 的引用
    int index = 0;              //开场 3 个动画帧的索引
    int status = -1;            //0 代表足球动画，1 代表背景转进来，2 代表全部渐
显，3 代表待命
    int alpha = 255;            //透明度，初始为 255，即不透明
    int [] layout = {3,3,4};    //玩家球员的站位数组，3 个值分别代表前场、中场、后场
    CustomGallery cg;           //自定义的 Gallery 类，用于选择俱乐部 logo
    Bitmap [] bmpLayout;        //代表前场、中场、后场 3 个阵线的图片数组
    Bitmap bmpPlus;             //加号图片
    Bitmap bmpMinus;            //减号图片
    Bitmap bmpPlayer;           //玩家图片
    Bitmap [] bmpSound;         //声音开关图片数组
    Bitmap bmpStart;            //开始按钮图片
    Bitmap bmpQuit;             //退出按钮图片
    Bitmap [] bmpGallery;       //存储 Gallery 对象要显示的内容
    Bitmap [] bmpAnimaition;    //存储欢迎动画帧的数组
    Bitmap bmpBack;             //背景图片
    Matrix matrix;              //Matrix 对象，用来旋转背景图
    //构造器：初始化成员变量
    public WelcomeView(FootballActivity father) {
        super(father);
        this.father = father;
        getHolder().addCallback(this);
        initBitmap(father);                         //初始化图片
        matrix = new Matrix();                      //创建 Matrix 对象
        cg = new CustomGallery(10,10,100,100); //创建 CustomGallery 对象
        cg.setContent(bmpGallery);       //为 CustomGallery 对象设置显示内容
```

```
        cg.setCurrent(2);                //设置 CustomGallery 当前显示的图片
        wt = new WelcomeThread(this);   //创建 WelcomeThread 对象
        wdt = new WelcomeDrawThread(this,getHolder());          //创建
WelcomeDrawThread 对象
        status = 0;                                         //设置初始状态值为 0
    }
    public void initBitmap(Context context){//初始化图片
        Resources r = context.getResources();      //获取 Resources 对象
        bmpBack = BitmapFactory.decodeResource(r, R.drawable.welcome);//
创建背景图片
        bmpLayout = new Bitmap[3];//创建表示前场、中场、后场的图片数组
       bmpLayout[0] = BitmapFactory.decodeResource(r, R.drawable.fwd field);
       bmpLayout[1] = BitmapFactory.decodeResource(r, R.drawable.mid field);
       bmpLayout[2] = BitmapFactory.decodeResource(r, R.drawable.bck field);
        bmpPlus = BitmapFactory.decodeResource(r, R.drawable.plus);//创建
加号图片
        bmpMinus = BitmapFactory.decodeResource(r, R.drawable.minus); //
创建减号图片
    bmpPlayer = BitmapFactory.decodeResource(r, R.drawable.player20);//创
建球员图片
        bmpSound = new Bitmap[2];                   //创建声音开关图片数组
       bmpSound[0] = BitmapFactory.decodeResource(r, R.drawable.sound1);

       bmpSound[1] = BitmapFactory.decodeResource(r, R.drawable.sound2);
         //创建开始图片按钮
bmpStart = BitmapFactory.decodeResource(r, R.drawable.start);
        //创建开始图片按钮
bmpQuit = BitmapFactory.decodeResource(r, R.drawable.quit);
        bmpAnimaition = new Bitmap[3];                       //创建动画数组
       bmpAnimaition[0] = BitmapFactory.decodeResource(r, R.drawable.p1);
       bmpAnimaition[1] = BitmapFactory.decodeResource(r, R.drawable.p2);
        bmpAnimaition[2] = BitmapFactory.decodeResource(r, R.drawable.p3);
        //初始化 Gallery 的图片资源
        bmpGallery = new Bitmap[8];     //创建自定义 Gallery 要显示的内容图片数组
        for(int i=0;i<bmpGallery.length;i++){
           bmpGallery[i] = BitmapFactory.decodeResource(r, father.imageIDs[i]);
        }
    }
    public void doDraw(Canvas canvas) {        //方法：用于根据不同状态绘制屏幕
        Paint paint = new Paint();           //创建画笔
        switch(status){
        case 0://显示 3 个动画帧
            canvas.drawBitmap(bmpAnimaition[index], 0, 0, null);
            break;
        case 1://背景图片旋转而进
            canvas.drawColor(Color.BLACK);     //清屏幕
            Bitmap bmpTemp = Bitmap.createBitmap(bmpBack, 0, 0,
                     //旋转背景图
bmpBack.getWidth(), bmpBack.getHeight(), matrix, true);
            canvas.drawBitmap(bmpTemp, 0, 0, null);            //绘制背景图
            break;
```



```
        case 2://全场透明
        case 3://全场待命--------------这两个画法一样，只是透明度不同
            canvas.drawColor(Color.BLACK);          //清屏幕
            paint.setAlpha(alpha);                  //设置透明度
            canvas.drawBitmap(bmpBack, 0, 0, paint);//画背景
            cg.drawGallery(canvas,paint);       //画自定义的 Gallery
            for(int i=0;i<layout.length;i++){   //对于球场上各个阵线上的信息进
行绘制
                //绘制阵线名称即前场、中场、后场
canvas.drawBitmap(bmpLayout[i], 0, 200+40*i, paint);
                for(int j=0;j<layout[i];j++){
canvas.drawBitmap(bmpPlayer, 65+j*18, 205+40*i, paint);    //根据阵线上人数
绘制球员
                }
                canvas.drawBitmap(bmpPlus, 244, 200+40*i, paint);  //绘制
加号按钮
                canvas.drawBitmap(bmpMinus, 280, 200+40*i, paint); //绘制
减号按钮
            }
            canvas.drawBitmap(bmpSound[father.wantSound?0:1], 135, 370,
paint); //绘制声音开关
            canvas.drawBitmap(bmpStart, 205, 425, paint);  //绘制开始按钮
            canvas.drawBitmap(bmpQuit, 25, 425, paint);
            //绘制退出按钮
            break;
        }
    }
    @Override
   public void surfaceChanged(SurfaceHolder holder, int format, int width,
            int height) {                       //重写 surfaceChanged 方法
    }
    @Override
    public void surfaceCreated(SurfaceHolder holder) {//重写 surfaceCreated 方法
        if(!wt.isAlive()){          //启动后台修改数据线程
            wt.start();
        }
        if(!wdt.isAlive()){         //启动后台绘制线程
            wdt.start();
        }
    }
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {   //重写
surfaceDestroyed 方法
        if(wt.isAlive()){                          //停止后台修改数据线程
            wt.isWelcoming = false;
        }
        if(wdt.isAlive()){                             //停止后台绘制线程
            wdt.flag = false;
        }
    }
}
```

(3) 文件 WelcomeThread.java

类 WelcomeThread 继承于 Thread 类，能够控制 WelcomeView 中的内容。具体实现代码如下。

```
/*
 * 该类继承自 Thread，主要实现欢迎界面的后台数据
 * 的修改以实现动画效果
 */
public class WelcomeThread extends Thread{
    WelcomeView father;              //WelcomeView 对象的引用
    boolean isWelcoming = false;     //线程执行标志位
    float rotateAngle = 60;          //旋转角度
    int rotateCounter = 0;           //旋转计数器
    int animationCounter=0;          //换帧计数器
   int sleepSpan = 150;              //休眠时间
    //构造器：初始化主要成员变量
    public WelcomeThread(WelcomeView father){
        this.father = father;
        isWelcoming = true;
    }
    public void run(){//线程的执行方法
        while(isWelcoming){
            switch(father.status){//获取现在的状态
            case 0:         //该状态为 3 个图片轮流显示
                animationCounter++;              //换帧计数器自加
                if(animationCounter == 4){       //计数器达到 4 时换帧
                    father.index ++;
                    if(father.index == 3){       //判断是否播放完毕所有帧
                        father.status = 1;       //转入下一状态
                    }
                    animationCounter = 0;        //清空计数器
                }
                break;
            case 1://该状态为背景图片旋转着进来
                father.matrix.postRotate(rotateAngle);      //旋转角度
                rotateCounter++;                            //计数器自加
                if(rotateCounter == 6){//旋转计数器到了
                    father.status = 2;//设置状态
                    father.alpha = 0;   //设置 alpha 值，用于菜单渐显
                }
                break;
            case 2://该状态为菜单渐显菜单渐显
                father.alpha +=51;          //alpha 值增加
                if(father.alpha >= 255){
                    father.status = 3;//进入待命状态，此状态玩家可以选择菜单选项
                }
                break;
            case 3: //如果遇到了待命状态，就自己把自己关闭
                this.isWelcoming = false;
                break;
```

```
            }
            try{
                Thread.sleep(sleepSpan );          //休眠一段时间
            }
            catch(Exception e){
                e.printStackTrace();               //捕获并打印异常
            }
        }
    }
}
```

(4) 文件 WelcomeDrawThread.java

类 WelcomeDrawThread 继承于 Thread 类，能够定时刷新 WelcomeView。具体实现代码如下。

```
/*
 * 该类继承自 Thread，主要负责定时刷新 WelcomeView
 */
public class WelcomeDrawThread extends Thread{
    WelcomeView father;                            //WelcomeView 对象的引用
    SurfaceHolder surfaceHolder;                   //WelcomeView 对象的
SurfaceHolder
    int sleepSpan = 100;                           //休眠时间
    boolean flag;                                  //线程执行标志位
    //构造器：初始化主要的成员变量
    public WelcomeDrawThread(WelcomeView father,SurfaceHolder surfaceHolder){
        this.father = father;
        this.surfaceHolder = surfaceHolder;
        this.flag = true;
    }
    //方法：线程执行方法
    public void run(){
        Canvas canvas = null;                      //创建一个 Canvas 对象
        while(flag){
            try{
                canvas = surfaceHolder.lockCanvas(null);   //为画布加锁
                synchronized(surfaceHolder){
                    father.doDraw(canvas);         //重新绘制屏幕
                }
            }
            catch(Exception e){
                e.printStackTrace();               //捕获异常并打印
            }
            finally{
                if(canvas != null){                //释放画布并将其传回
                    surfaceHolder.unlockCanvasAndPost(canvas);
                }
            }
            try{
                Thread.sleep(sleepSpan);           //休眠一段时间
            }
```

```
            catch(Exception e){
                e.printStackTrace();            //捕获异常并打印
            }
        }
    }
}
```

12.6.3 加载节目

加载界面在游戏开发项目中比较常见，是整个游戏表示层中比较简单的部分，主要涉及了类 LoadingView，类 LoadingView 继承于 SurfaceView 类，并且实现了 SurfaceHolder.Callback 接口。具体实现代码如下。

```
/*
 * 该类继承自 SurfaceView，主要的功能是在后台加载、创建对象时在前台 显示进度
 */
public class LoadingView extends SurfaceView implements
SurfaceHolder.Callback{
    FootballActivity father;        //Activity 的引用
    Bitmap bmpProgress;             //显示进度时图片
    Bitmap [] bmpProgSign;          //进度条上的标志物
    Bitmap bmpLoad;                 //进度条图片对象
    int progress=0;                 //进度，0 到 100
    int progY = 330;                //进度条的 Y 坐标
    LoadingDrawThread lt;           //LoadingView 的刷屏线程
    public LoadingView(FootballActivity father) {//构造器，初始化主要成员变量
        super(father);          //调用父类构造器
        this.father = father;
        initBitmap(father);     //初始化图片
        getHolder().addCallback(this); //添加 Callback 接口
        lt = new LoadingDrawThread(this,getHolder());//创建刷屏线程
    }
    public void doDraw(Canvas canvas) {//方法：绘制屏幕
        canvas.drawColor(Color.BLACK);                  //清屏幕
        canvas.drawBitmap(bmpLoad, 10, 100, null);      //画加载时图片
        canvas.drawBitmap(bmpProgress, 5, progY, null);    //画进度条图片
        //画遮盖物
        Paint p = new Paint();                          //创建画笔对象
        p.setColor(Color.BLACK);                        //设置画笔颜色
        int temp = (int)((progress/100.0)*320);     //将进度值换算成屏幕上的长度
        canvas.drawRect(temp, progY, 315, progY+20, p);    //画遮盖物挡住进
度条图片
        //画进度条标志物
        for(int i=0;i<3;i++){
            canvas.drawBitmap(bmpProgSign[i], 140*i, progY-10, null);
        }
        if(progress == 100){                    //绘制进度条已满的提示文字
            p.setTextSize(13.5f);
            p.setColor(Color.GREEN);
            canvas.drawText("单击屏幕开始游戏...", 100, progY+50, p);
```

```
        }else{                                    //绘制进度条未满的提示文字
            p.setTextSize(13.5f);
            p.setColor(Color.RED);
            canvas.drawText("加载中,请稍后....", 120, progY+50, p);
        }
    }
    public void initBitmap(Context context){//方法：初始化图片
        Resources r = context.getResources();      //获取资源对象
        //初始化进度条图片
bmpProgress = BitmapFactory.decodeResource(r, R.drawable.progress);
        bmpProgSign = new Bitmap[3];                    //初始化进度条标志物
        bmpProgSign[0] = BitmapFactory.decodeResource(r, R.drawable.prog1);
        bmpProgSign[1] = BitmapFactory.decodeResource(r, R.drawable.prog2);
        bmpProgSign[2] = BitmapFactory.decodeResource(r, R.drawable.prog3);
        bmpLoad = BitmapFactory.decodeResource(r, R.drawable.load);//初始
化加载图片
    }
    @Override
    protected void finalize() throws Throwable {
       System.out.println("############ LoadingView  is dead##########");
        super.finalize();
    }
    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int
width,int height) {//重写 surfaceChanged 方法
    }
    @Override
    public void surfaceCreated(SurfaceHolder holder) {//重写 surfaceCreated 方法
        if(!lt.isAlive()){          //如果后台刷屏线程还未启动，就启动线程刷屏
            lt.start();
        }
    }
    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {//重写 surfaceDestroyed 方法
        lt.flag = false;            //停止刷屏线程
    }
}
```

12.6.4 运动控制

运动控制模块用于控制足球和玩家运动模块的开发，主要涉及了类 Ball、PlayerMoveThread、AIThread 和 Player。

(1) 球员控制模块

球员控制功能是通过按键上的方向键实现的，这是在文件 FootballActivity.java 中定义的，在里面定义了 onKeyDown(int keyCode, KeyEvent event)和 onKeyUp(int keyCode, KeyEvent event)，分别处理键盘按下事件和键盘抬起事件。对应代码如下。

```
    public boolean onKeyDown(int keyCode, KeyEvent event) {//处理键盘按下事
件的回调方法
```

```
        switch(keyCode){
        case 21:                        //左
            keyState = keyState | 2;
            keyState = keyState & 0xfffffffe;       //清除掉其他的键盘状态
            break;
        case 22:                        //右
            keyState = keyState | 1;
            keyState = keyState & 0xfffffffd;       //清除掉其他的键盘状态
            break;
        default:
            break;
        }
        return true;
    }
    @Override
    public boolean onKeyUp(int keyCode, KeyEvent event) {  //处理键盘抬起事
件的回调方法
        switch(keyCode){
        case 21:                        //左
            keyState = keyState & 0xfffffffd;       //清楚该状态位
            break;
        case 22:                        //右
            keyState = keyState & 0xfffffffe;       //清楚该状态位
            break;
        default:
            break;
        }
        return true;
    }
```

(2) 控制手机一方球员的运动

手机一方球员的运动控制是通过一个算法实现的，算法如下。

在每个固定时间读取足球的运动方向，如果足球偏向左，则将球员运动方向设置为向左；如果足球偏向右，则将球员运动方向设置为向右。上述功能是通过 AIThread 类实现的，此类为 AI 的后台线程，实现的功能是让 AI 足球运动员根据足球的运动参数来确定自己的运动方向(向左还是右)。由于人工智能算法比较复杂，本书也不是专门用来讲解算法原理的，所以采用了一个比较简单的算法，即如果足球的方向偏左(即可以是左上、左下、正左等，共 7 个方向)，那么 AI 的运动方向就是向左，反之则向右。

实现文件 AIThread.java 的对应代码如下。

```
public class AIThread extends Thread{
    GameView father;                //视图类引用
    boolean flag;                   //循环控制变量
    int sleepSpan = 30;             //睡眠时间
    //构造器,初始化成员变量
    public AIThread(GameView father){
        this.father = father;
        flag = true;    //设置线程标志位
    }
```

```
    //线程启动后的执行方法
    public void run(){
        while(flag){
            int d = father.ball.direction;      //获取足球运动方向
            if(d >0  && d<8){                   //如果足球方向偏左
                father.aiDirection = 4;         //AI 运动方向改为向左
            }
            else if(d>8  && d<15){              //如果足球方向偏右
                father.aiDirection = 12;        //AI 运动方向改为向右
            }
            try{
                Thread.sleep(sleepSpan);        //休眠一段时间
            }
            catch(Exception e){
                e.printStackTrace();            //打印并捕获异常
            }
        }
    }
}
```

(3) PlayerMoveThread 类

前面介绍了玩家和球员的移动，最终目的是实现双方球员位置变化的是 PlayerMoveThread 线程。此线程的实现文件是 PlayerMoveThread.java，具体代码如下。

```
public class PlayerMoveThread extends Thread{
    FootballActivity father;    //Activity 的引用
    boolean outerFlag;          //线程执行标志位
    boolean flag;               //是否需要移动球员的位置标志位
    int sleepSpan = 20;         //线程休眠时间
    boolean myMoving;       //为 true 表示玩家可移动，为 false 表示玩家不可动
    boolean aiMoving;       //为 true 表示 AI 可移动，为 false 表示 AI 不可动
    //构造器，初始化主要成员变量
    public PlayerMoveThread(FootballActivity father){
        super.setName("##-PlayerMoveThread");  //为线程设置名称，调试使用
        this.father = father;
        outerFlag = true;
        flag = true;
        myMoving = true;            //初始状态玩家的球员是可移动的
        aiMoving = true;            //初始状态电脑 AI 的球员是可移动的
    }
    //方法：线程的执行方法
    public void run(){
        while(outerFlag){
            while(flag){
                //修改玩家和 AI 运动员的位置
                if(father.current == father.gv){ //如果 FieldView 是当前屏幕
                    if(myMoving){                   //如果玩家的球员是可移动的
                        int key = father.keyState; //读取键盘监听状态
                        if((key & 1) == 1){         //键盘状态为向右
            father.gv.movePlayers(father.gv.alMyPlayer, 4);//调用方法向右移
动球员
```

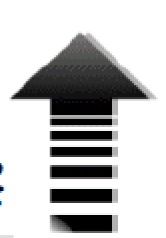
```
                        }
                        else if((key & 2) == 2){    //键盘状态为向左
            father.gv.movePlayers(father.gv.alMyPlayer, 12);   //调用方法向
左移动球员
                        }
                        else{                      //将 direction 设置为静止-1
                            father.gv.movePlayers(father.gv.alMyPlayer, -1);
                        }
                    }
                    if(aiMoving){                        //判断 AI 是否可以移动
                        int d = father.gv.aiDirection;      //读取 AI 球员的
运动方向
                father.gv.movePlayers(father.gv.alAIPlayer, d);//修改 AI 运
动员的位置
                    }
                }
                try{
                    Thread.sleep(sleepSpan);        //线程休眠一段时间
                }
                catch(Exception e){
                    e.printStackTrace();            //打印并捕获异常
                }
            }
            try{
                Thread.sleep(300);  //当不需要移动玩家时，线程空转后睡眠一段时间
            }
            catch(Exception e){
                e.printStackTrace();        //捕获并打印异常
            }
        }
    }
}
```

(4) Player 类

在本项目中，每个 Player 对象代表一个球员，在 Player 中除了成员变量之外，还有一个成员方法。并且还包括一个成员方法 levelUp，负责在玩家赢得一场比赛的胜利之后升级的操作。Player 在文件 Player.java 中定义，具体代码如下。

```
public class Player{
    int x;                              //球员中心的 x 坐标
    int y;                              //球员中心的 y 坐标
    int movingDirection=-1;             //运动员的运动方向，12 左 4 右
    int movingSpan = 1;                 //移动步进
    int attackDirection;                //进攻方向，0 上 8 下
    int power=10;                       //踢球时给球的速度大小

    public void levelUp(){              //升级后调用
        movingSpan+=1;                  //每次移动的步进增大
        if(movingSpan > 5){
            movingSpan = 5;
        }
```



```
    }
}
```

(5) Ball 类

Ball 类是一个继承自 Thread 类的线程类，是系统中比较重要的类，在里面不仅封装了足球对象的必要信息，还提供了用于碰撞检测和处理的一系列相关方法，Ball 类是整个项目后台运行的核心。在本项目中，设置了足球的移动方向有 16 个，夹角是 22.5 度。Ball 类是在文件 Ball.java 中定义的，下面开始介绍其实现流程。

第一步：是封装足球有关的信息，如坐标点、方向、移动速度等。定义成员变量，具体代码如下。

```
public class Ball extends Thread{
    int x;                              //足球中心的 x 坐标
    int y;                              //足球中心的 y 坐标
    int direction=-1;                   //足球的运动方向，从 0 到 15 顺时针代表从向上
开始的 16 个方向，写书的时候画个图贴上去
    int velocity=20;                    //足球的运动速率
    int maxVelocity = 20;               //最大运动速率
    int minVelocity = 5;                //最小运动速率
    int ballSize = 10;                  //足球大小
    Matrix matrix;                      //Matrix 对象，用来实现足球图片的翻转效果
    Bitmap bmpBall;                     //足球的图片
    GameView father;                    //FieldView 对象引用
    float acceleration=-0.10f;          //足球在无人撞击时速度会逐渐衰减
    boolean isStarted;                  //比赛是否开始
    boolean isPlaying;                  //比赛是否正在进行
    float sin675=0.92f;                 //特定角度正弦值，用于计算移动的像素个数
    float sin225=0.38f;                 //特定角度正弦值，用于计算移动的像素个数
    float sin45=0.7f;                   //特定角度正弦值，用于计算移动的像素个数
    int sleepSpan = 50;                 //休眠时间
    float changeOdd = 0.6f;             //变向的概率
    int lastKicker;                     //最近的这一脚是谁踢的，0 代表自己，8 代表 AI
```

第二步：实现游戏碰撞检测处理。此功能主要是通过 run 方法来实现的，run 方法中主要有两个方法 move 和 checkCollision。前者负责根据足球的方向(16 种之一)来的移动足球的位置。后者用于进行碰撞检测，查看是否足球碰到 AI 或玩家的运动员，是否遇到边界，是否遇到一些 Bonus 如冰冻小球等。具体代码如下。

```
    //线程的任务方法
    public void run(){
        while(isStarted){
            while(isPlaying){
                //移动足球
                move();
                //碰撞检测
                checkCollision();
                //休眠一下
                try{
                    Thread.sleep(sleepSpan);
```

```
                }
                catch(Exception e){
                    e.printStackTrace();
                }
            }
            try{
                Thread.sleep(500);
            }
            catch(Exception e){     e.printStackTrace();
            }
        }
    }

    //移动足球
    public void move(){
        switch(direction){
        case 0:         //方向向上
            y -= velocity;                                          //移动
            decreamentVelocity();                                   //衰减速度
            break;
        case 1:         //上偏右 22.5 度
            x += (int)(velocity*sin225);                            //移动
            y -= (int)(velocity*sin675);
            decreamentVelocity();                                   //衰减速度
            break;
        case 2:         //上偏右 45 度
            x += (int)(velocity*sin45);                             //移动
            y -= (int)(velocity*sin45);
            decreamentVelocity();                                   //衰减速度
            break;
        case 3:         //上偏右 67.5 度
            x += (int)(velocity*sin225);                            //移动
            y -= (int)(velocity*sin675);
            decreamentVelocity();                                   //衰减速度
            break;
        case 4:         //方向向右
            x += velocity;                                          //移动
            decreamentVelocity();                                   //衰减速度
            break;
        case 5:         //右偏下 22.5 度
            x += (int)(velocity*sin675);                            //移动
            y += (int)(velocity*sin225);
            decreamentVelocity();                                   //衰减速度
            break;
        case 6:         //右偏下 45 度
            x += (int)(velocity*sin45);                             //移动
            y += (int)(velocity*sin45);
            decreamentVelocity();                                   //衰减速度
            break;
        case 7:         //右偏下 67.5 度
            x += (int)(velocity*sin225);                            //移动
```

```
            y += (int)(velocity*sin675);
            decreamentVelocity();                               //衰减速度
            break;
        case 8:          //方向向下
            y += velocity;                                      //移动
            decreamentVelocity();                               //衰减速度
            break;
        case 9:          //下偏左 22.5 度
            x -= (int)(velocity*sin225);                        //移动
            y += (int)(velocity*sin675);
            decreamentVelocity();                               //衰减速度
            break;
        case 10:         //下偏左 45 度
            x -= (int)(velocity*sin45);                         //移动
            y += (int)(velocity*sin45);
            decreamentVelocity();                               //衰减速度
            break;
        case 11:         //下偏左 67.5 度
            x -= (int)(velocity*sin675);                        //移动
            y += (int)(velocity*sin225);
            decreamentVelocity();                               //衰减速度
            break;
        case 12:         //方向向左
            x -= velocity;                                      //移动
            decreamentVelocity();                               //衰减速度
            break;
        case 13:         //左偏上 22.5 度
            x -= (int)(velocity*sin675);                        //移动
            y -= (int)(velocity*sin225);
            decreamentVelocity();                               //衰减速度
            break;
        case 14:         //左偏上 45 度
            x -= (int)(velocity*sin45);                         //移动
            y -= (int)(velocity*sin45);
            decreamentVelocity();                               //衰减速度
            break;
        case 15:         //左偏上 67.5 度
            x -= (int)(velocity*sin225);                        //移动
            y -= (int)(velocity*sin675);
            decreamentVelocity();                               //衰减速度
            break;
        default:
            break;
        }
    }
    public void checkForBorders(){
        int d = direction;
        //左右是不是出边界了
        if(x <= father.fieldLeft){
            //撞了左边界
            if(d>8 && d<16 && d!=12){   //如果不是正撞到左边界
```




```
                if(Math.random() < changeOdd){   //一定概率沿正确反射路线变向
                    direction = 16 - direction;
                }
                else{                            //一定概率随机变向
                    direction = (direction>12?1:5) +
(int)(Math.random()*100)%3;
                }
            }
            else if(d == 12){            //如果是正撞到左边界
        if(Math.random() < 0.4){//注意这个概率要小，因为正撞上去希望随机变向的概
率大一些
                    direction = 4;
                }
                else{
                    direction = (Math.random() > 0.5?3:5);
                }
            }
        }
        else if(x > father.fieldRight){
            //撞到右边界
            if(d >0 && d<8 && d!=4){
                if(Math.random() < changeOdd){      //按正常反射路线变向
                    direction = 16-direction;
                }
                else{                               //一定概率随机变向
                   direction = (direction>4?9:13) + (int)(Math.random()*100)%3;
                }
            }
            else if(d == 4){             //如果是正撞到右边界
                if(Math.random() < 0.4){
                    direction = 12;
                }
                else{
                    direction = (Math.random()>0.5?11:13);
                }
            }
        }
        d = direction;
        //判断是否撞到上边界
        if(y < father.fieldUp){
            //不是正撞
            if(d>0 && d<4 || d>12&&d<16){
                if(Math.random() < changeOdd){    //一定概率沿正确反射路线变向
                    direction = (d>12?24:8) - d;
                }
                else{                                   //一定概率随机变向
                    direction = (d>12?9:5) + (int)(Math.random()*100)%3;
                }
            }
            else if(d == 0){             //正撞到上边界
                if(Math.random() < 0.4){         //一定概率沿正确反射路线返回
```

```
                        direction = 8;
                    }
                    else{
                        direction = (Math.random() < 0.5?7:9);    //一定概率随机变向
                    }
                }
            }
            //判断是否撞到下边界
            else if(y > father.fieldDown){
                //不是正撞
                if(d >4 && d<12 && d!=8){
                    if(Math.random() < changeOdd){ //按正常反射路线变向
                        direction = (d>8?24:8) - d;
                    }
                    else{                           //随机变向
                        direction = (d>8?13:1) +(int)(Math.random()*100)%3;
                    }
                }
                else if(d == 8){        //正撞到下边界
                    if(Math.random() < 0.4){        //正常变向
                        direction = 0;
                    }
                    else{                           //随机变向
                        direction = (Math.random()>0.5?1:15);
                    }
                }
            }
        }
        /*
         * 此方法检测是否碰到手机运动员，如果碰到，则调用 handleCollision 方法处理碰撞，
         * 同时播放声音设置足球新速率和设置 lastKicker
         */
        public void checkForAIPlayers(){
            int r = (this.ballSize + father.playerSize)/2;
            for(Player p:father.alAIPlayer){
                if((p.x - this.x)*(p.x - this.x) + (p.y - this.y)*(p.y -
this.y) <= r*r){        //发生碰撞
                    handleCollision(this,p);            //处理碰撞
                    if(father.father.wantSound &&
father.father.mpKick!=null){    //播放声音
                        try {                           //用 try/catch 语句包装
                            father.father.mpKick.start();
                        } catch (Exception e) {}
                    }
                    velocity = p.power;
                    lastKicker = 8;                 //记录最后一脚是谁踢的
                }
            }
        }
        /*
         * 此方法检测是否碰到了玩家的足球运动员，
```

```
     */
    public void checkForUserPlayers(){
        int r = (this.ballSize + father.playerSize)/2;
        for(Player p:father.alMyPlayer){
            if((p.x - this.x)*(p.x - this.x) + (p.y - this.y)*(p.y -
this.y) <= r*r){        //发生碰撞
                handleCollision(this,p);            //处理碰撞
                if(father.father.wantSound &&
father.father.mpKick!=null){    //播放声音
                    try {
                        father.father.mpKick.start();
                    } catch (Exception e) {}
                }
                velocity = p.power;          //被赋予新速度
                lastKicker = 0;              //记录最后一脚谁踢的
            }
        }
    }
```

在上述代码中，方法 checkForBorders()能够检测足球是否碰到了边界，如果碰到了上下左右中的某一边界，则处理方法为：一定概率沿正确的路线(类似反射定律)改变方向，一定概率随机变向，如以方向 1 碰到上边界，会以某个较大的概率将小球的方向改为 7，而会有相对较小的概率将方向改为 5、6、7 三个方向中的某一个。

第三步：定义方法 handleCollision()，此方法处理足球和运动员之间的碰撞，手机和玩家的处理方式是一样的，首先查看 Player 对象的 movingDirection，再综合 Player 对象的 attackDirection，确定方向范围，类似直角坐标系中的 4 个象限，然后在方向范围中随机产生一个，这样产生的方向有惯性在里面，这样看来会比较真实。需要注意的是，如果足球和运动员碰撞时运动员静止不动，那么可选的方向就是 1 或 15(进攻方向朝上)、7 或 9(进攻方向朝下)。具体代码如下。

```
    public void handleCollision(Ball ball,Player p){
        switch(p.movingDirection){
        case 12:                    //移动方向向左
            if(p.attackDirection == 0){     //攻击方向向上
                ball.direction = 13 + (int)(Math.random()*100)%3;//取 13、
14、15 中一个
            }
            else{                           //攻击方向向下
                ball.direction = 9 + (int)(Math.random()*100)%3;   //取
9、10、11 中一个
            }
            break;
        case 4:                     //移动方向向右
            if(p.attackDirection == 0){     //攻击方向向上
                ball.direction = 1 + (int)(Math.random()*100)%3;   //取
1、2、3 中一个
            }
            else{                           //攻击方向向下
                ball.direction = 5 + (int)(Math.random()*100)%3;   //取
```

```
5、6、7 中一个
            }
            break;
        default:                //没有移动
            if(p.attackDirection == 0){     //攻击方向向上
                ball.direction = 15 + (int)(Math.random()*100)%3;//取 1、
2、3 中一个
                if(ball.direction > 15){
                    ball.direction = ball.direction % 16;
                }
            }
            else{                           //攻击方向向下
                ball.direction = 7 + (int)(Math.random()*100)%3;//取 7、
8、9 中一个
            }
            break;
        }
    }
```

第四步：定义方法 checkIfScoreAGoal()，此方法用于检测是否进球，如果是，则相应球队得分加 1，然后判断游戏是否结束(游戏规则是谁先进够 8 个谁就赢)。具体代码如下。

```
    public void checkIfScoreAGoal(){
        if(this.y <= father.fieldUp && this.x > father.AIGoalLeft &&
this.x < father.AIGoalRight){
            //上方球门进球,即玩家
            isPlaying = false;
            father.scores[0]++;
            father.checkIfLevelUp();
        }
        else if(this.y >= father.fieldDown && this.x > father.myGoalLeft
&& this.x < father.myGoalRight){
            //AI 进球
            isPlaying = false;
            father.scores[1]++;
            father.checkIfLevelUp();
        }
    }
```

12.6.5　奖品模块

本项目的奖品模块功能是通过类 Bonus 实现的，它是 IceBonus 和 LargerGoalBonus 的父类，主要提供一些公共的成员或是方法。一个 Bonus 类主要包括自己的绘制部分、触发后的绘制、自己生命周期计时、触发后生命周期计时、触发后后台数据的修改、触发生命周期结束后后台数据的恢复。Bonus 类是在文件 Bonus.java 中实现的，具体代码如下。

```
public abstract class Bonus{
    public static final int PREPARE = 0;    //准备态，可以画出来，但是不可以碰到
    public static final int LIVE = 1;       //活动态，可以被画出，可以被碰到
    public static final int DEAD = 2;   //死亡态，不可以被画出，不可以被碰到
```




```
    public static final int EFFECTIVE = 3;//生效态，不可以被画出，不可以被碰
到，但是可以画其产生的作用，如冰冻等
    public static final int LIFE SPAN = 5000;
    public static final int EFFECT SPAN = 5000;
    public static final int PREPARE SPAN = 2000;
    int status = -1;            //0：存在，1：死亡，2：生效
    int x,y;                    //Bonus 中心点的坐标
    int bonusSize;              //Bonus 的大小
    int selfIndex=0;            //自己帧索引
    int effectIndex = 0;        //生效后的索引
    int selfFrameNumber;        //自己动画帧总数
    int effectFrameNumber;      //生效动画帧总数
    int target;                 //对谁起作用 0 为自己，8 为 AI，以他们的进攻方向区分

    GameView father;            //FieldView 对象引用

    Bitmap [] bmpSelf;          //用于绘制自己的 Bitmap 数组
    Bitmap [] bmpEffect;        //用于绘制效果的 Bitmap 数组

    Timer timer = new Timer();      //创建定时器对象
    List<Bonus> owner;              //记录自己被添加到哪个集合中
    //设置一段时间后才可以从 PREPARE 到 LIVE 态
    public void setPrepareTimeout(int timeout){
        timer = new Timer();
        timer.schedule(new TimerTask(){
            @Override
            public void run() {
                Bonus.this.status = Bonus.LIVE;
                setTimeout(Bonus.LIFE SPAN);
            }
        },
        timeout);
    }
    //设置定时的方法
    public void setTimeout(int timeout){
        timer = new Timer();
        timer.schedule(new TimerTask(){     //调用成员方法 schedule 来启动一个
定时器
            @Override
            public void run() {
                Bonus.this.status = Bonus.DEAD;         //杀死 Bonus
                Bonus.this.undoJob();                   //调用 undoJob 方法
                Bonus.this.owner.remove(Bonus.this);    //从其所属的集合中移
除自己
                father.balDelete.add(Bonus.this);   //将自己添加到待删除集合中
            }
        },
        timeout);
    }
    //画自己的方法
    public void drawSelf(Canvas canvas){
```

```
        canvas.drawBitmap(bmpSelf[(selfIndex++)%selfFrameNumber], x-
bonusSize/2, y-bonusSize/2, null);
    }
    //抽象方法：绘制效果，需要子类重写
    public abstract void drawEffect(Canvas canvas);
    //抽象方法：修改后台数据，需要子类重写
    public abstract void doJob();
    //抽象方法：恢复后台数据，需要子类重写
    public abstract void undoJob();
    //抽象方法：设置目标，需要子类重写
    public abstract void setTarget(int lastKicker);
}
```

BonusManager 类继承自 Thread，主要实现对 Bonus 对象的管理，通过定期检测屏幕上 Bonus 的个数，根据概率随机生成 Bonus。文件 BonusManager.java 的实现代码如下。

```
public class BonusManager extends Thread{
    boolean flag = false;       //设置线程执行标志位
    GameView father;            //FieldView 对象引用
    int sleepSpan = 3000;       //休眠时间
    int maxBonus = 2;       //设置最大的 Bonus 个数
    //构造器，初始化主要成员变量
    public BonusManager(GameView father){
        this.father = father;
        this.flag = true;
    }
    //方法：线程的 run 方法
    public void run(){
        while(flag){
            //只在游戏正常状态下才产生 Bonus
            if((!father.isGameOver) && (!father.isScoredAGoal)
&&(!father.isShowDialog)){
                generateBonus();        //调用 generateBonus 方法产生 Bonus
            }
            try{
                Thread.sleep(sleepSpan);    //休眠一段时间
            }
            catch(Exception e){
                e.printStackTrace();        //打印捕获异常
            }
        }
    }
    //随机生成 Bonus
    public void generateBonus(){
        int currentBonusNumber = father.balLive.size();    //获取活着的
Bonus 的个数
        float generateOdd = 1 - currentBonusNumber*0.33f;  //计算生成概率
        if(Math.random() < generateOdd){    //产生一个随机数，如果小于生成概率
            int x = (int)(Math.random() * (father.fieldRight-
father.fieldLeft))+ father.fieldLeft;
            int y = (int)(Math.random() * (father.fieldDown -
```

```
father.fieldUp)) + father.fieldUp;
            Bonus b;
            if(Math.random() > 0.5){     //产生随机数，如果值小于0.5，则创建IceBonus
                b = new IceBonus(father,x,y);
            }
            else{                       //如果随机数小于0.5，则创建LargerGoalBonus
                b = new LargerGoalBonus(father,x,y);
            }
            b.status = Bonus.PREPARE;   //设置状态为准备态
            father.balLive.add(b);      //将Bonus添加到用于碰撞检测的集合中
            b.owner = father.balLive;   //设置其所属者
            father.balAdd.add(b);       //将Bonus添加到待添加集合中

            b.setPrepareTimeout(Bonus.PREPARE_SPAN);    //为刚生成的Bonus设
置准备超时
        }
    }
}
```

至此，整个项目的主要模块功能介绍完毕，执行之后的初始界面如图 12-5 所示；菜单界面如图 12-6 所示；进度条界面如图 12-7 所示；游戏界面如图 12-8 所示。

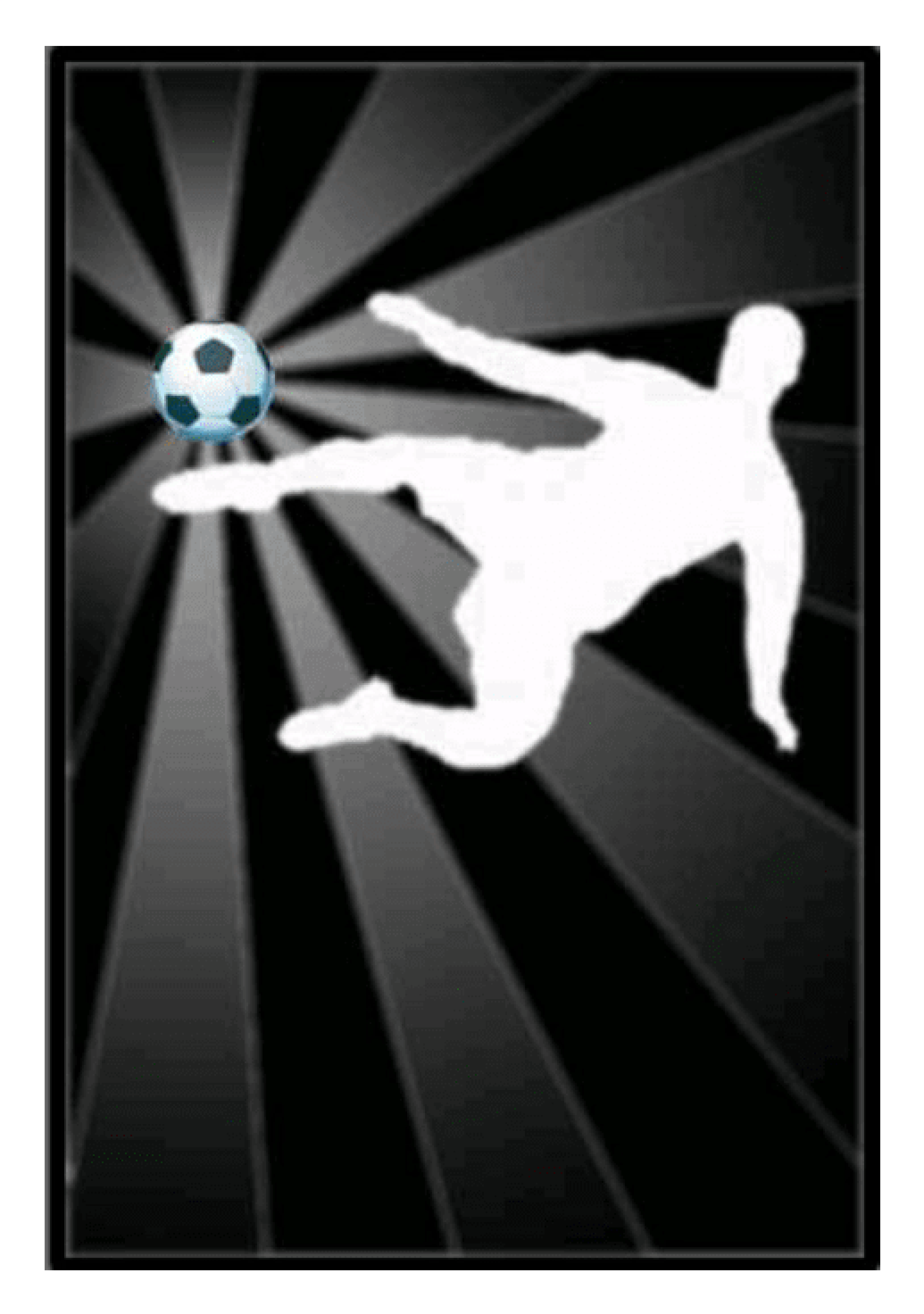


图 12-5　初始界面

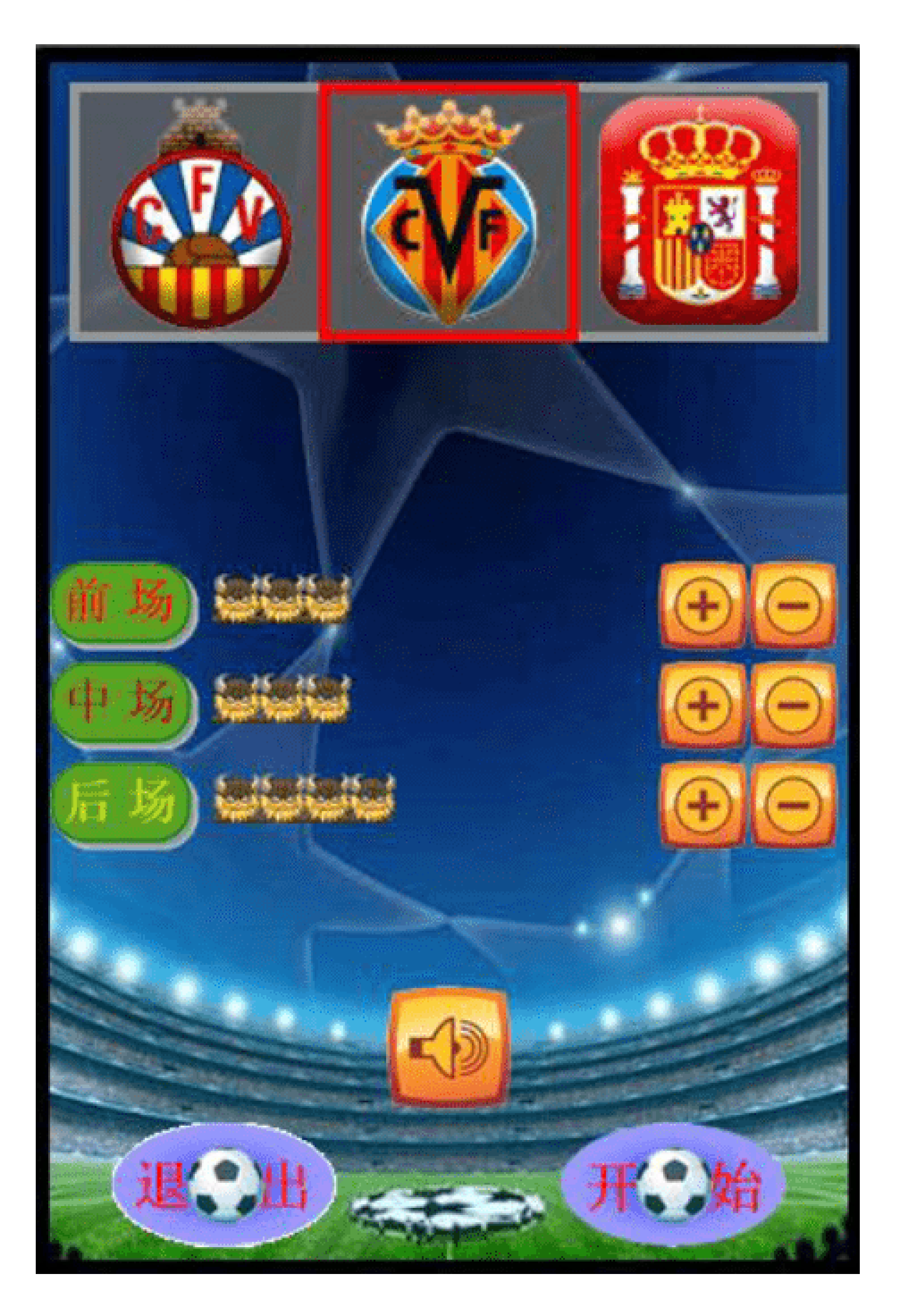


图 12-6　菜单界面

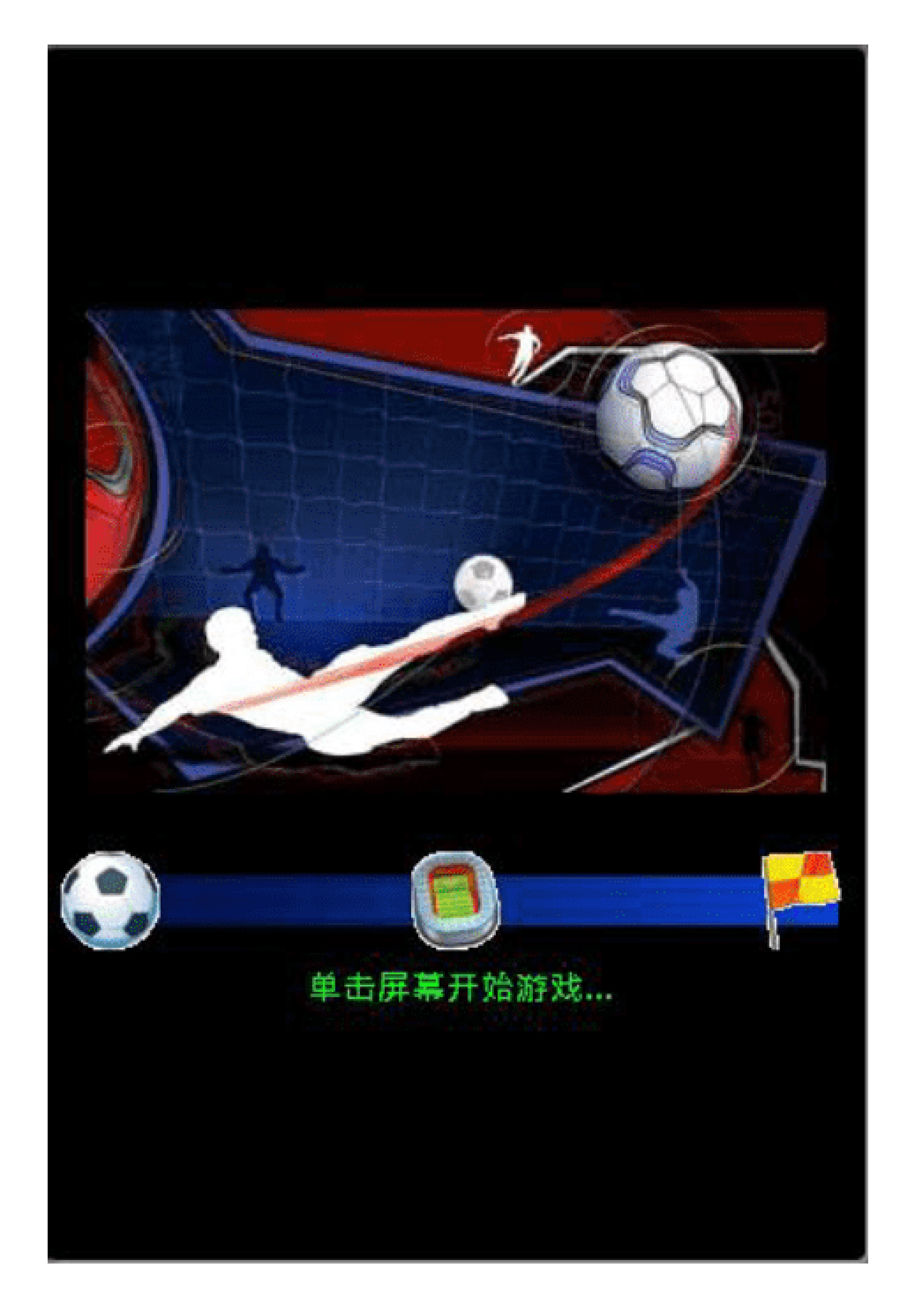


图 12-7　进度条界面



图 12-8　游戏界面